VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Paralelní výpočty na Windows HPC pomocí MPI

Parallel Computing on Windows HPC Using MPI

2010

Lukáš Vojáček

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 5. května 2010 .....

Rád bych na tomto místě poděkoval všem, kteří mi s prací pomohli, protože bez nich by tato práce nevznikla.

Abstrakt

Tato práce se zabývá využitím MPI na Microsoft HPC v programovacích jazycích C++ a C#. Popisuje způsoby jakými lze MPI aplikace spouštět na Windows HPC Serveru 2008. Dalším krokem je porovnání MPI funkcí použitých v jazycích C++ a C#. MPI je dále využita pro paralelizaci dvou neuronových sítí, Backpropagation a Self-organizing map. U každé neuronové sítě jsou provedeny testy na učinnost jejich paralelizace.

Klíčová slova: MPI, Windows HPC, Backpropagation, SOM

Abstract

This work deals with the use of MPI for Microsoft HPC programming languages C++ and C#. This work describes methods by which are possible to run MPI applications on the Windows HPC Server 2008. The next step is to compare the MPI functions used in C++ and C#. MPI is also used for the parallelization of two neural networks, Backpropagation and Self-organizing map. For each neural network are tested for their force the parallelization.

Keywords: MPI, Windows HPC, Backpropagation, SOM

Seznam použitých zkratek a symbolů

MPI	–	Message Passing Interface
HPC	–	High Performance Computing
SOM	–	Self-organizing map
OOSol	–	Object Oriented SOLvers

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

V dnešní době se snažíme co nejvíce snížit čas potřebný na výpočet velkých úloh. Jednou z možností jak tohoto cíle dosáhnout je rozdělit úlohu na menší části a ty pak dále paralelně počítat na různých výpočetních strojích. Protože jednotlivé části musí spolu komunikovat, tak pro tento účel byla vytvořena specifikace MPI[12]. Microsoft vydal operační systém Windows HPC Server 2008[16], který se právě zabývá paralelnímy výpočty. Zároven vydal i svoji implementaci MPI, která má řadu vylepšení.

Cílem této diplomové práce bylo, abychom se naučli vytvářet MPI aplikace (což zahrnuje ladění i trasování), dále způsoby jakým lze spouštět tyto MPI aplikace na Windows HPC Serveru 2008 a celkově se naučli pracovat s tímto operačním systémem.

Nasbíraných znalostí jsme poté využili k napsání dvou neuronových sítí. Vybrali jsme dvě nejznámější neuronové sítě a to Backpropagation (učení s učitelem) a Kohoneovu mapu (učení bez učitele). Samotný průchod neuronovou sítí, kde se přivedou na vstup hodnoty poté očekáváme jen výstup, tak je rychlý, ale oproti tomu učení může být zdlouhavý proces, který jsme chtěli urychlit.

Testovat naše aplikace máme možnost na školním Windows HPC Serveru 2008.

Dále spolupracujeme na projektu jehož cílem je vytvořit paralelní algoritmy pro urychlení matematických výpočtů.

1.1 Struktura práce

V tomto textu je celkově šest kapitol, kde první je úvod. Ve druhé kapitole se zabýváme Windows HPC Serverem 2008, kde popisujeme samotný systém a jeho vlastnosti (Sekce 2.1). Následovně popisujeme job (Sekce 2.2) a jeho jednotlivé fáze. Abychom mohli psát programy spustitelné na Windows HPC Serveru 2008, je nutné projekt správně nastavit (Sekce 2.3) a jak je vůbec spustit (Sekce 2.4). Při hledání chyb potřebujeme často ladit program (Sekce 2.5).

Ve třetí kapitole popisujeme základní informace o MPI, jak vypadá napsaný MPI program v jazycích C++ (Sekce 3.1) a C# (Sekce 3.2). Dále porovnáváme rychlosti přenosu pomoci MPI (Sekce 3.3).

Ve čtvrté kapitole popisujeme neuronové sítě a naši implementaci Backpropagation (Sekce 4.1) a Kohonenovu mapu (Sekce 4.2). Zároveň jsme provedli testy jejich rychlostí.

Naše spolupráce na vývoji distribuovaných datových struktur je popsána v páté kapitole.

Zhodnocené výsledky jsou uvedeny v závěru (Kapitola 6). Následuje přehled literatury a uživatelské příručky.

2 Windows HPC Server

Dokument popisující Microsoft Windows HPC Server 2008 a ze kterého je pro tuto kapitolu čerpáno je White Paper [16] od Microsoftu. Microsoft Windows HPC Server 2008 je novou generaci High Performance Computing. Nabízí kompletní a integrované clustrové prostředí včetně operačního systému, Job Scheduleru (popsáno dále v kapitole), podporu Message Passing Interface v2 (MPI2[12]), správu clusteru a monitorování jednotlivých komponent. Windows HPC Server 2008 postaven na systému Windows Server 2008 64-bit, může efektivně škálovat pro tisíce procesorových jader a obsahuje konzoli pro správu, která pomáhá monitorovat a udržovat systém. Plánování úloh umožňuje integraci mezi Windows a na Linuxu založených HPC platformách. Dále podporuje dávkové zatížení a zatížení pomocí servisně orientované architektury (SOA).

Windows HPC Server 2008 může být propojen s ostatními produkty od Microsoftu, aby pomohl zvýšit produktivitu HPC. Mezi ně patří například Microsoft Office SharePoint Server 2007[17], Windows Workflow Foundation[18] nebo System Center 2007[19]. Díky integraci s Windows Communication Foundation (WCF)[23], Windows HPC Server 2008 umožňuje vývojářům pracovat se SOA aplikacemi a využít sílu paralelních výpočtů, které nabízí HPC.

Při vývoji OS Windows HPC Server 2008 se Microsoft zaměřil na čtyři klíčové oblasti: systémy řízení, plánování úloh, práce se sítí a MPI a poslední je správa dat. Některé z výhod jsou popsány následovně:

Systémy řízení

- Seskupení uzlů umožňuje správcům kategorizovat a dávkovat operace, které budou spuštěny na výpočetních uzlech
- Vylepšení výpočetních uzlů pomocí Windows Deployment Services[20]
- Přidána diagnostika systému

Plánování úloh

- Umožňuje přidělení úlohy na jádro procesoru, socket procesoru nebo na samotný výpočetní uzel
- Byli přidány nové politiky přidělování úloh
- Vylepšení výkonu pro velké clustery

Sítě a MPI

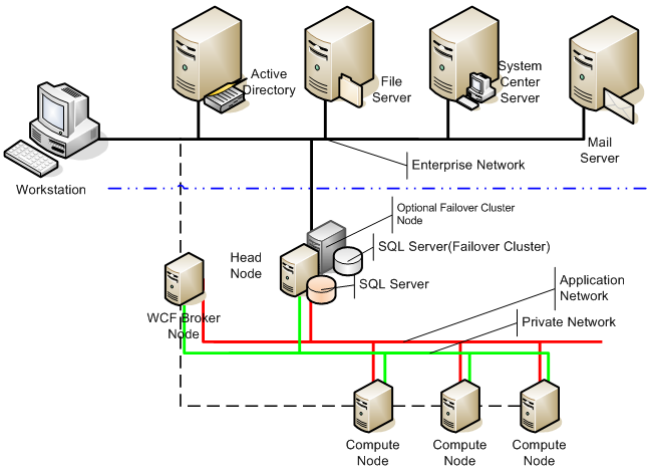
- Přidáno nové síťové rozhraní Remote Direct Memory Access, díky kterému jsou aplikace psané pomoci MPI mnohem výkonnější
- Implementace MS MPI má novou sdílenou paměť pro více jádrové servery

Správa dat

- Zdokonalen Server Message Block v2[14]

Architektura

Windows HPC Server 2008 je složen z několika (dílčích) serverů, které obsahují jeden hlavní uzel a jeden nebo více výpočetních uzlů. Hlavní uzel poskytuje vysokou dostupnost služeb, SQL Server[21], zprostředkovává veškerý přístupu ke zdrojům clusteru a je jediný kdo může řídit a plánovat úlohy na výpočetních clustech. Windowsh HPC Server 2008 umožňuje využívat služby Active Directory[22] pro zabezpečení, správu účtů a celkově pro řízení provozu pomocí nástrojů jako jsou System Center Operations Manager. Na obrázku 1 je možné vidět ukázkové zapojení.



Obrázek 1: Ukázka zapojení Windows HPC Server 2008 (převzato z [16], stránka 5)

Síťové zapojení

Windows HPC Server 2008 podporuje pět různých síťových zapojení, tak aby uživatel měl co největší možnost výběru toho co mu vyhovuje. Tato zapojení jsou následující:

1. Výpočetní uzly jsou připojeny pouze do soukromé sítě. Na hlavním uzlu jsou dvě síťové karty, kde jedna je připojena do soukromé sítě a druhá do podnikové sítě. Aby se výpočetní uzly mohly připojit také do podnikové sítě, musí být na hlavním uzlu povolen NAT (network addres translation[13]).

2. Všechny uzly jsou připojeny jak do soukromé sítě, tak do veřejné. Každý uzel má dvě síťové karty, kde jednu vyžívá k přístupu do podnikové sítě a druhou do soukromé sítě.

3. Výpočetní uzly jsou připojeny pouze do soukromé a aplikační sítě. Na hlavním uzlu jsou tři síťové karty, kde jedna je připojena do soukromé sítě, druhá do podnikové sítě a třetí do aplikační sítě. Stejně jako v bodě 1, hlavní uzel využívá NAT aby umožnil výpočetním uzlům přístup do podnikové sítě. Výpočetní uzly mají dvě síťové karty, kde jednu vyžívá k přístupu do soukromé sítě a druhou do aplikační sítě sítě (obvykle vysoko rychlostní síť).

4. Všechny uzly jsou připojeny do soukromé, veřejné i aplikační sítě. Každý uzel má tři síťové karty, kde jednu vyžívá k přístupu do podnikové sítě, druhou do soukromé sítě a třetí do aplikační sítě.

5. Všechny uzly jsou připojeny pouze do podnikové sítě. Toto propojení má nevýhodu, že každý výpočetní uzel musí být samostatně nainstalován.

Poznámka 2.1 Školní Windows HPC server je zapojen podle první topologie. Kde je povolen jak NAT, tak i nakonfigurován DHCP server pro soukromou síť, která má rychlost 1 Gbps.

2.1 Job

Informace o tom, co to je job a jak se ním pracuje jsme získali z White Paperu [15], kde jsou uvedeny další informace. Jedná se o důležitou část Windows HPC, kde se job se může skládat z jednoho tasku a nebo může obsahovat těchto tasku více. Taskem je myšleno spustitelný program, ať už standardní (sekvenční) a nebo paralelní (např. MPI, WCF). Task je možno spustit také interaktivně jako SOA aplikaci. Každý task je přidělen do skupiny (Group), těchto skupin může být více (defaultně je vytvořena pouze jedna skupina, ve které jsou všechny tasky). Tasky v jednotlivých skupinách jsou spuštěny až se ukončí tasky v předchozí skupině. Každému jobu je možné přidělit uzel na kterém má běžet a také mu speciálně přidělit prostředky (paměť, procesory).

Job Scheduler je další důležitou částí, protože se stará o samotné joby (tasky). Mezi jeho povinnosti patří přidělovat zdroje pro výpočet, spouštět tasky na výpočetních uzlech, sledovat stav v jakém se joby, tasky a výpočetní uzly nacházejí. Umožňuje dále například při chybovém provedení jobu(tasku) automaticky znovu jej spustit.

Windows HPC Server 2008 podporuje dva typy tasku, jedním je základní (práce sním je ukázána v podkapitole 2.3) a druhým je parametrický task.

- **Základní task** používá jeden příkazový řádek, který obsahuje příkaz pro spuštění a metadata, která popisují jak daný příkaz spustit. Může se jednat o paralelní task a může běžet na více uzlech nebo jádrech. Paralelní tasky obvykle spolu komunikují pomoci MS-MPI nebo prostřednictvím sdílené paměti, za předpokladu že běží na jednom uzlu.
- **Parametrický task** se liší od předchozího tím, že příkazový řádek obsahuje proměnné, díky čemuž je možné spouštět ten samý úkol vícekrát s různou hodnotou u proměnných. Jinak jej také lze spouštět paralelně.

Od vytvoření až po ukončení prochází job postupně třemi fázemi:

1. **Odeslání**
2. **Plánování**
3. **Spuštění**

2.1.1 Odeslání

Jedná se o první fázi ve které se nejprve musí samotný job vytvořit a poté jej odeslat na zpracování. Existuje několik možností jak tohoto docílit a asi nejjednodušší jsou Command Line Interface, Windows Powershell, HPC Pack 2008 Job Management Console nebo napsání vlastní aplikace (HPC Pack 2008 API umožňuje pracovat s Job Scheduler).

Existuje velké množství parametru, které se dají u jobu nastavit. Pro ukázku jsme vybrali ty nejzajímavější jako je priorita, doba běhu jobu, počet přidělených procesorů/uzlů/socketu, velikost paměti, určení uzlů kterých má job využívat.

Toto byly parametry jobu, každý task má ovšem ještě dodatečné parametry. Tyto se vztahují na samotný program, jeho umístění, adresář kde budou standardní vstupy, výstupy a chyby (při spuštění MPI aplikace na Windows HPC server 2008, jsou standardní vstupy přesměrovány ze souboru a standardní výstupy a chyby do souboru) a počet procesorů/ uzlů/ socketu na kterých má běžet. Ukázku jak vytvářet joby a tasky najdete dále v této kapitole.

2.1.2 Plánování

Tato fáze má nastarosti spravování a přidělování zdrojů k jednotlivým jobům. Existuje několik druhů politik, podle kterých přiděluje jednotlivé zdroje. K základním politikám Priority-based FCFS, Backfilling a Nonexclusive schedules, byly přidány ve Windows HPC Server 2008 ještě následující: Resource matchmaking, Job Templates, MCRA (Multi-Level Compute Resource Allocation), Grow/shrink a Preemption.

Priority-Based FCFS

Tato plánovací politika spojuje dva přístupy, jedním je metoda kdo dříve přijde dříve bude obsloužen a druhá záleží na prioritě, s jakou byl job spuštěn. Všechny joby jsou seřazeny ve frontě podle času odeslání a jejich spuštění je prováděno od nejvyšší priority (Highest) po nejmenší (Lowest).

Backfill

Uživatele při odeslání jobu nastaví, jak dlouho může být job spuštěn a jaké zdroje jsou požadovány. Tyto zdroje jsou rezervovány na základě odeslání jobu. Jestliže Job Scheduler nalezne volné zdroje na tak dlouho, jak je požadováno u nějakého "malého" jobu, tak jej může spustit. Díky tomuto se mohou "malé" joby vykonat dříve a nemusí čekat ve frontě, až na ně dojde řada a zároveň se zvýší využitelnost clusteru.

Exclusive Scheduling

Pokud nastavíme job jako exkluzivní, pak žádný jiný job nemůže být spuštěn na daném uzlu. To samé platí i o tasku, tudíž pouze daný task může běžet na uzlu.

Resource Matchmaking

Uživatel určí jaké jsou požadované výpočetní, síťové a aplikační zdroje a Job Scheduler umístí job tam, kde jsou tyto požadované zdroje dostupné. U toho přístupu může dojít k plýtvání zdrojů. Příklad špatného využití je např. exkluzivní spuštění aplikace, která vyžaduje 4 procesory, ale je spuštěná na 8 procesorech.

Job Templates

Správce vytvoří šablony pro joby, kde přesně určí jaké zdroje jsou potřeba pro dané procesy a nastaví prioritu pro uživatelské skupiny. Šablona obvykle definuje nastavení jobu pro danou aplikaci a uživatel, když jej používá, nemusí vědět jak se joby nastavuji, ale stačí mu pouze vědět, jak jej odeslat ke zpracování.

Protože správce přiděluje jednotlivým skupinám prioritu s jakou se mají spouštět joby, tak dokáže ovlivňovat druh výpočtu jaký se bude na clusteru provádět.

MCRA

Tento typ politiky umožňuje zpracovávat joby na základě granularity s nimiž bude naplánováno jak využívat výpočetních zdrojů. Zde závisí jestli se job bude spouštět na jádrech, soketech nebo uzlech,kde pro každou variantu je lepší jiný druh aplikace. Např. sokety jsou vhodné pro aplikace, které málo přistupují k paměti.

Grow/Shrink

Tato politika pracuje s joby, které mají více tasků a principem je měnit za běhu jobu zdroje, které jsou mu přiděleny. Když jsou výpočetní zdroje dostupné můžou být přidány k jobu a naopak, když není task aktivní tak jsou zdroje odebrány. Opět se zvýší využitelnost clusteru a navíc se zajišťuje lepší rozvržení zdrojů na joby s nejvyšší prioritou.

Preemption

Principem této politiky je odebrat přidělené zdroje jobům s nízkou prioritou a přidělit je jobům, které mají nejvyšší prioritu. Cílem je snížit čas potřebný k výpočtu u preferovaného jobu. Jsou popsány dva režimy:

1. **Immediate Preemption**, kde joby s nízkou prioritou jsou ukončeny, aby uvolnili prostředky pro job s vysokou prioritou.

2. **Graceful Preemption**, kde jobu s nízkou prioritou se odebere část jeho zdrojů ve prospěch jobu s vysokou prioritou. Sice bude výpočet pomalejší než pomocí prvního režimu, ale na druhou stranu žádný job nebude ukončen a tím se neztratí data na kterých pracuje.

Standardní přidělování zdrojů

Tato strategie závisí na následujících položkách: na počtu procesorů, požadovaných uzly pro joby, exkluzivní přidělení jobu, na přidělení uzlů k danému tasku a exkluzivní přidělení tasku.

Job Scheduler setřídí vhodné uzly podle paměti a rychlosti. Kde uzly jsou nejdříve setřízeny podle velikosti paměti a poté teprve podle rychlosti. Toto je základní část, kterou obsahují všechny strategie, ale je možné ji přizpůsobit. V následujícím kroku se přidělí procesory ze setřízených uzlů a to podle nastavených hodnot počtu procesorů: minimální a maximální.

Pokud jsou vybrány uzly na kterých se má job spustit, pak Job Scheduler nesetřízuje všechny uzly, ale pouze přiděluje procesory z vybraného seznamu. Opět při přidělování procesorů se řídí minimálním a maximálním počtem, kolik jich bylo nastaveno při spuštění.

Pro vhodné rozdělování zdrojů na Windows HPC Serveru 2008, by se uživatel měl řídit následujícími doporučeními ohledně vytváření, odesílaní a spouštění jobu.

1. Nenechávat job spuštěný bez časového omezení (což je základní nastavení), ale pokaždé nastavit nejdelší možný čas, po kterém by úloha měla být dokončena.

2. Speciálně přiřazovat uzly k jobům, jen v tom případě, že daný uzel má zdroje, které jsou nutné pro daný job.

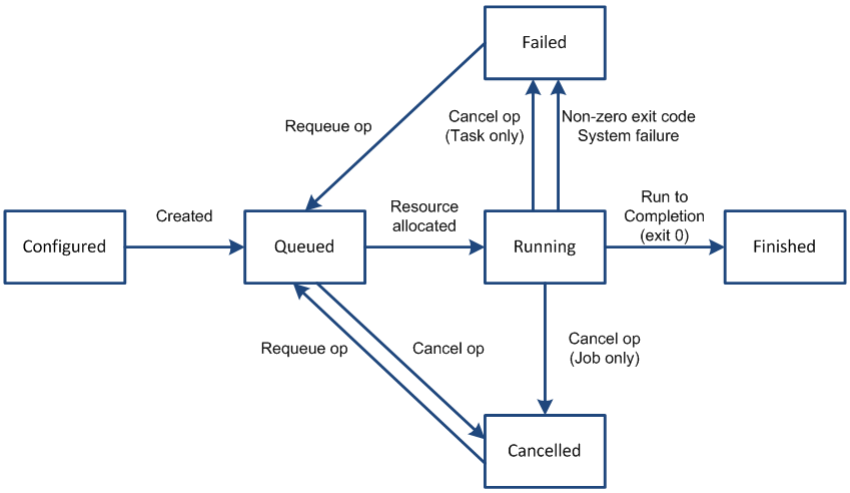
3. Při zadávání počtu procesorů v jobu, je vhodné nastavit minimální počet procesorů na hodnotu, která je co nejmenší přijatelná pro výpočet. Protože pokud by minimální hodnota byla nastavena na ideální pro výpočet, job muže čekat než mu budou přiděleny dostatečné zdroje.

4. Pro každý job přidělit příslušný počet uzlů. Pokud se spustí job na dvou uzlech, ale pro výpočet je nutný pouze jeden uzel, tak zbytečně je rezervován jeden uzel.

5. Exkluzivní spouštění jobů nebo tasků provádět jen v případě, že je to nezbytně nutné.

2.1.3 Spuštění

Poslední co zbývá je samotné spuštění jobu. Job i task se mohou nacházet v jednom z šesti možných stavů (configuring, queued, running, finished, failed, cancelled). Ukázka jak task prochází jednotlivými fázemi je znázorněno na obrázku 2.



Obrázek 2: Stavy ve kterých se task může nacházet (převzato z [12], stránka 23)

Správci mohou kontrolovat joby pomoci filtrů, ve kterých jsou nastaveny podmínky, které musí job splnit. Existují dva druhy filtrů: pro odeslání jobu a pro spuštění jobu.

Možnosti použití filrů:

- Kontrola, zda job není spuštěn bez časového omezení
- Ověření, zda název projektu je platný a zda uživatel, který jej chce odeslat má k tomuto úkolu oprávnění

2.2 Kompilace

Pro používání MPI (popis naleznete v kapitole 3) funkcí ve Visual Studio 2008, je nutné upravit projekt. Tato úprava se liší podle použitého programovacího jazyka.

Poznámka 2.2 Předpokládá se že je nainstalován na počítači alespoň *HPC Pack 2008 SDK* pro C++, pro C# je nutné ještě nainstalovat MPI.NET[11] (Jedná se o implementaci se kterou jsme pracovali a není závazná).

- V jazyce C# stačí v projektu přidat reference na *Message Passing Interface*. Postup je převzat z [10] a je následující:

  1. Vybereme *Project→Add Reference*

2. Vybereme záložku *.NET*
3. Přidáme *Message Passing Interface*

- V jazyce C++ je to trochu složitější, je nutné přidat cesty na adresářům kde se nachází hlavičkový soubor *mpi.h* a knihovna *msmpi.lib*. Postup je vytvořen na základě [8] a je následující:

  1. Vybereme *Project→<Název projektu> Properties*
  2. Vybereme *Configuration Properties*
  3. Přidání hlavičkového souboru se provede v *C/C++ → General* v kolonce *Additional Include Directories*, zde zadá cesta k souboru *mpi.h*. Jako příklad uvedu cestu: C:\Program Files\Microsoft HPC Pack 2008 SDK\Include .
  4. Přidání knihovny má dvě části. Zadá se adresář kde se nachází a to *Linker → General* v kolonce *Additional Library Directories*. Zde záleží jestli se používá architektura 32 bit nebo 64 bit. Příklad pro 32 bit: C:\Program Files\Microsoft HPC Pack 2008 SDK\Lib\i386. A druhá část přidání názvu knihovny v *Linker → Input* v kolonce *Additional Dependencies*, kde se zadá *msmpi.lib*.

     Poznámka 2.3 Je důležité si zjistit jestli uzly podporují obě kompilace Debuge i Release a nebo pouze Release.

  5. Někdy i po tomto nastavení nemusí jít spustit MPI aplikace na uzlech, a ještě se musí upravit v *C/C++ →Code Generation* položka *Runtime Library* na **Multi-threaded Debug (/MTd)**. Na doprovodném CD je program z názvem Testy-PrenosuCpp a ten obsahuje kompletní spustitelné nastavení.

Po této upravě je možné využívat funkce MPI.
U Debuge kompilace nedoporučujeme mít, při psaní MPI aplikace v jazyce C++, nastaveno v konfiguraci projektu *Configuration Properties→Linker→General* hodnotu u proměnné *Target Machine* na *MachineX64 (/MACHINE:X64)*. Je to z jednoduchého důvodu a to když se použije příkaz **assert** v programu a hodnota která vstupuje do něho je nepravdivá (false), tak se ukončí všechny procesy kde je tento příkaz. Problém je takový, že v *Job Manageru* se takový job jeví jako že běží (je ve stavu Runnig) a jedním možným způsobem jak zjistit že program se ukončil, je připojit se na daný uzel a podívat se kolik procesů tam skutečně je spuštěných. Nebo se podívat do souboru kam se ukládají chyby. Tento soubor bude mít sice velikost 0, ale při otevření zde bude nějaký výpis. Ukázka zdrojového kódu, který způsobí výše popsaný problém, je zobrazen ve výpisu 1.

```
#include <assert.h>
#include "mpi.h"

int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    int myrank;
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    if (myrank!=1)      //pouze proces s rankem 1 nevola funkci assert, toto je jen ukazka
    {
        assert(false);
    }
    MPI_Finalize();
    return 0;
}
```

Výpis 1: Ukázka programu způsobující chybu v C++

2.3 Spuštění

Paralelní program můžeme spustit buď na skutečných jádrech procesorů nebo jej lze spustit jako více procesů na jednom nebo více procesorů. Při vývoji aplikace je vhodné využívat druhou metodu a to z několika důvodů.

- Zbytečně nevyužívat zdroje, zvlášť pokud více uživatelů využívá Windows HPC Server.
- Pokud by byly všechny zdroje využity, musíte čekat než se váš program dostane na řadu.
- Převážně se spouští na uzlech už Release verze.
- Aplikace je možné vyvíjet i mimo Windows HPC Server 2008, např. na vlastním počítači.

Jednou z nejdůležitějších částí programování je ladění (debug) programu. Je možné používat ladící výpisy, což je někdy nepraktické nebo použit Visual studio, které se ovšem musí řádně nastavit. Toto nastavení se pro jazyky C++ a C# trochu liší a kompletní je popsáno dále v této kapitole.

Spuštění procesů na lokálním počítači

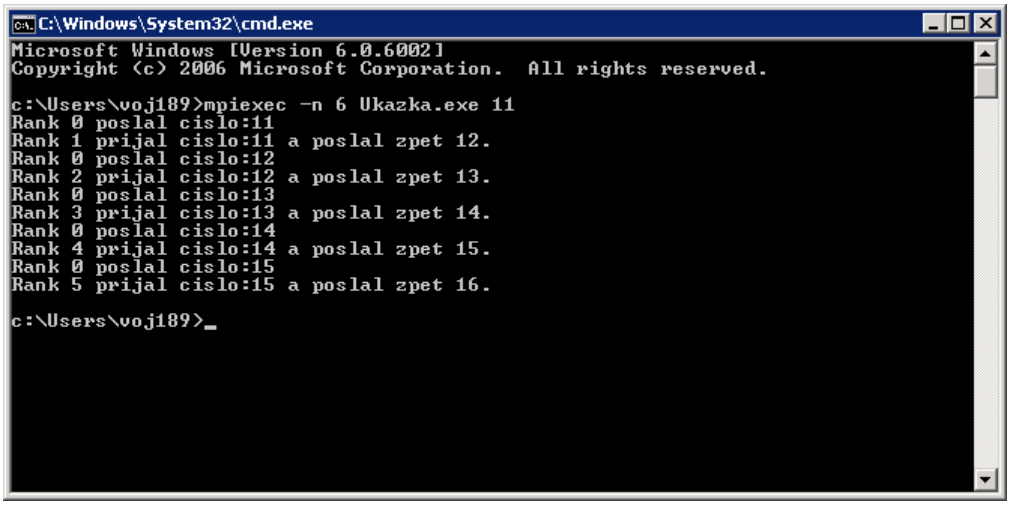
Pro spuštění programu se využívá aplikace *mpiexec.exe*.

Kde v příkazovém řádku se zadá: C:\Program Files\Microsoft HPC Pack 2008 SDK\Bin\ mpiexec.exe -n 4 Ukazka.exe

Kde parametrem *n* určujeme počet procesorů a poté název souboru který chceme spustit. Ukázka použitého příkazu lze vidět na obrázku 3, kde je také přidán parametr číslo 11, což je stejné jako spouštění normálního programu s parametrem. Kompletní výpis možnosti získáte při použití parametru *help2*.

Spuštění na uzlech

Spuštění programu na více jádrech je možné rozdělit do dvou kategorii a to pokud chceme daný program spouštět lokálně a nebo vzdáleně přes síť. Pokud bychom chtěli lokálně je nutné na každém uzlu mít stejné umístění programu např. C:\Ukazka\Ukazka.exe.



Obrázek 3: Použití příkazu mpiexec

Tohoto je možné docílit příkazem *clusrun copy* (Pro spuštění příkazu *clusrun* je nutné mít administrátorské oprávnění) nebo jej ručně rozkopírovat. Při vzdáleném spuštění je stačí pouze nechat sdílet daný program.

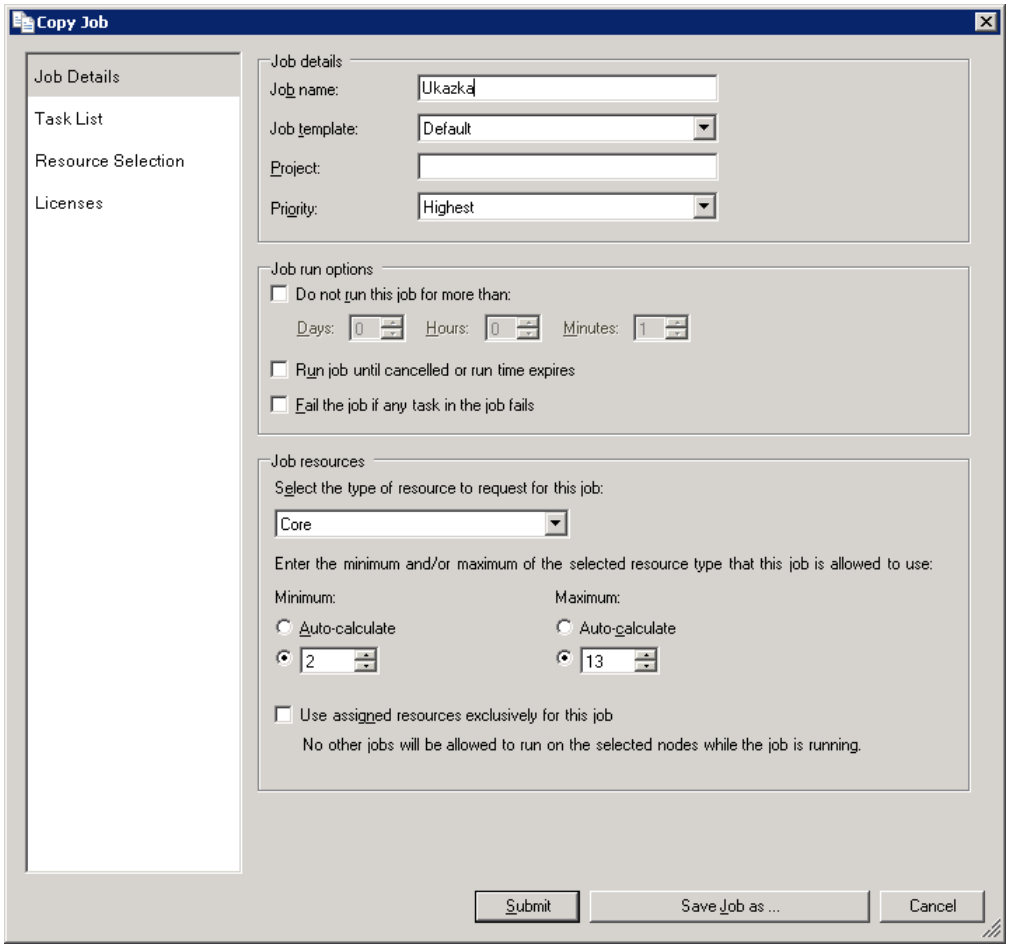
Pro vytvoření jobu je nejjednodušší způsob grafické rozhraní (alespoň pro začátečníky) a to se docílí spuštěním aplikace **HPC Job Manager**. Kde v pravém menu je výběr *New Job*. Zobrazí se okno jaké lze vidět na obrázku 4. V této záložce se nastaví základní parametry jobu, jako je např. název, druh a počet zdrojů, priorita. V další záložce (Task List) se přidávají jednotlivé tasky k tomuto jobu. Na obrázku 5 je vidět vytvořený jeden task, který obsahuje název, soubor který se má paralelně spustit (opět se spouští pomocí souboru mpiexec), umístění souboru (lokální nebo vzdálený přístup), standardní vstupy, výstupy a chyby (důležitou věcí je že standardní I/O je přesměrován na práci se soubory).

Poznámka 2.4 Pokud se výstup nastaví do sdíleného souboru, každý procesor bude ukládat data do toho souboru principem, kdo dříve přijde, dříve zapíše.

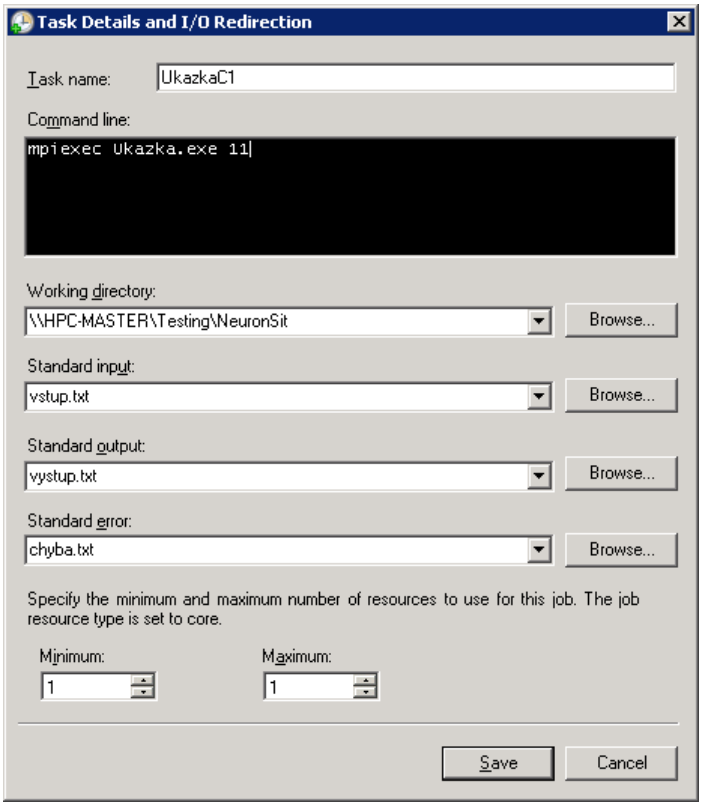
Poslední věcí je nastavení minimálního a maximálního počtu procesorů na kterém se má task spustit. Ve třetí záložce (Resource Selection) je možné vybrat uzly se kterými bude job pracovat a hardwarové prostředky jako je např. velikost paměti. Nakonec stačí job odeslat a počkat dokud se nezpracuje.

Další možností je vytvoření jobu z příkazové řádky, ukázku je možné vidět na obrázku 6. Pokud chceme at' se vykonává pouze jeden program, stačí vytvořit task a odeslat jej (ve skutečnosti se opět vytvoří job, ale pro uživatele je to jednodušší cesta pro zpracování jobu) jak lze vidět na obrázku 7.

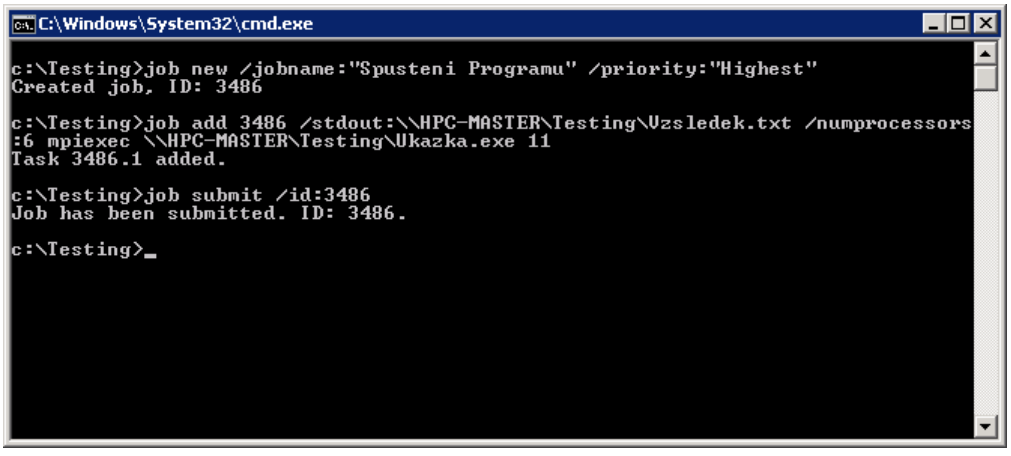
Při problému s práci s joby doporučujeme navštívit fórum, kde se tomuto problému věnují a to na adrese <http://social.microsoft.com/Forums/cs-CZ/windowshpcsched/threads>



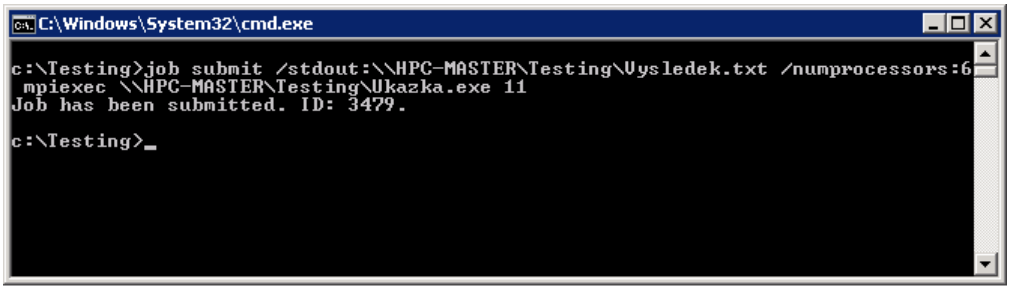
Obrázek 4: Vytvoření jobu pomocí HPC Job Manager



Obrázek 5: Vytvoření tasku pomocí HPC Job Manager



Obrázek 6: Vytvoření a odeslání jobu pomocí příkazové řádky



Obrázek 7: Vytvoření a odeslání tasku pomocí příkazové řádky

2.4 Ladění programu

Ladění programu ve Visual C#

Pokud chceme MPI program spouštěn pouze jako jeden proces, stačí standardně spustit *Debuge→Start Debugging*. U více procesů už se musí upravit nastavení projektu. Pro jazyk C# je to trochu složitější, protože Visual Studio 2008 nepodporuje možnost, že by se v jazyce C# psali MPI aplikace. Jde to však obejít a to následujícím způsobem (oproti dokumentu ze kterého jsme čerpali [2], je nutná jedna úprava).

1. Otevřeme nastavení projektu *Project→<Název projektu> Properties*
2. Vybereme záložku *Build*
3. Musí se nastavit platforma pro jakou je program určen. Defaultní nastavení je *Any CPU*, změnte toto nastavení buď na *x86* nebo *x64*.
4. Dalším krokem je přeložit takto nastavený program a spustit další Visual Studio 2008.
5. Nyní v novém Visual Studiu otevřeme přeložený exe soubor pomocí (File→Open→Project/Solution) a soubor se zdrojovým kódem, který chceme ladit. Tento soubor otevřeme pomocí (File→Open→File).
6. Další kroky jsou podobné jako při použití Visual C++. Otevřeme nastavení projektu *Project→Properties*
7. V ComboBoxu *Debugger to launch* vybereme *MPI Cluster Debugger*
8. Zde je nutné nastavit čtyři hodnoty, nejdříve u *MPIRun Command* nastavíme hodnotu *mpiexec* a u *MPIRun Arguments* nastavíme hodnotu na *-n X*, kde X je počet procesů (jedná se o standardní spuštění MPI aplikace). Poslední co je nutné nastavit je cesta k souboru *mpishim.exe* (Tento soubor není součástí MS MPI, ale standardně Visual Studia. Ještě je nutné rozlišit jestli chceme 32-bit a nebo 64 bit). Toto se nastaví u *MPIShim Location*, ukázka umístění: C:\Program Files\Microsoft Visual Studio 9.0\Common7\IDE\Remote Debugger\x86\ mpishim.exe. Ještě musíme změnit hodnotu *Auto* na *Mix*, bez této úpravy se program nezastaví na breakpointu.

Nyní stačí přidat breakpoint do zdrojového kódu a klávesou F5 spustit. Chování je totožné jako u Visual C++.

Ladění programu ve Visual C++

Pro jazyk C++ je to nastavení o trochu jednodušší, ale hodně podobné.

1. Otevřeme nastavení projektu *Project→<Název projektu> Properties*
2. Vybereme *Configuration Properties*
3. Vybereme *Debugging*
4. V ComboBoxu *Debugger to launch* vybereme *MPI Cluster Debugger*
5. Zde je nutné nastavit tři hodnoty, nejdříve u *MPIRun Command* nastavíme hodnotu *mpiexec* a u *MPIRun Arguments* nastavíme hodnotu na *-n X*, kde X je počet procesů (jedná se o standardní spuštění MPI aplikace). Poslední co je nutné nastavit je cesta k souboru *mpishim.exe* (Tento soubor není součástí MS MPI, ale standardně Visual Studia. Ještě je nutné rozlišit jestli chceme 32-bit a nebo 64 bit). Toto se nastaví u *MPIShim Location*, ukázka umístění: C:\Program Files\Microsoft Visual Studio 9.0\Common7\IDE\Remote Debugger\x86\ mpishim.exe.

Celkové nastavení je možné vidět na obrázku 8. S více procesy se při ladění pracuje téměř totožně jako s jedním procesem, akorát je potřeba otevřít si ve Visual Studiu okno *Processes* (Debuge→Windows→Processes), aby bylo možné mezi jednotlivými procesy přepínat.

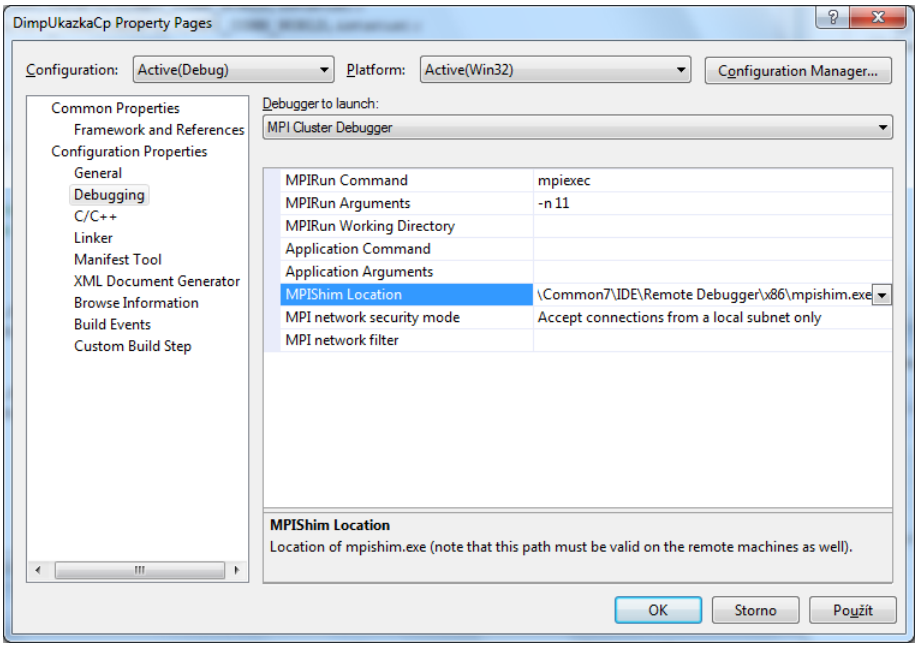
Trasování

Sepsali jsme návod podle videa [3] a je následující:

Pro vytvoření záznamu je nutné přidat, při spouštění MPI aplikace, pouze parametr *trace*.Data se budou ukládat do souboru umístěného v %USERPROFILE%\ mpi_trace.etl, toto umítění je možné změnit parametrem *tracefile*. Po ukončení programu jsou v souboru mpi_trace.etl uloženy veškeré informace přenosu, bohužel mohou být s rozdílným časem, tudíž je nutné je synchnonizovat a to programem *mpicsync* (je součástí MS MPI). Příklad: *mpicsync mpi_trace.etl*. Pro další použití je nutné změnit formát záznamu a to pomocí programu *etl2clog* (je součástí MS MPI). Příklad: *etl2clog mpi_trace.etl*. Nyní stačí jen zobrazit data, je možné použít např. aplikací napsanou v Jave *Jumpshot*. Jedná se volně stažitelnou aplikaci, kde je možné zobrazit si např. jen některé funkce, pouze určitý časový úsek, atd. Stručný postup:

1. mpiexec -trace
2. mpicsync
3. etl2clog
4. Jumpshot

Mnoho informací týkajících se trasování lze nalézt v dokumentu [4].



Obrázek 8: Nastavení projektu v jazyce C++

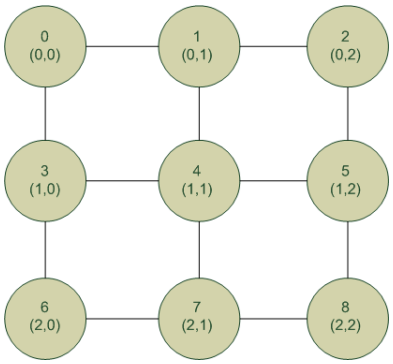
3 MPI

MPI popisuje jak se mají data přenášet mezi procesy. Nejedná se o implementaci, ale pouze o specifikaci. Veškeré operace jsou popsány jako funkce nebo metody v jazycích C, C++, Fortran77 a Fortran90. Informace ohledně specifikace MPI specifikací jsem čerpal z dokumentu o MPI2.2[12] a MS MPI z White paper [5].

Výhody MPI standartu jsou přenositelnost, flexibilita, snadné použití. Některé cíle, kterých autoři MPI chtěli dosáhnout:

- Jednoduché použíti v jazycich C, C++, Fortran 77 a Fortran 90.
- Efektivní komunikace mezi procesy
- Nezávislost na platformě
- Použití v heterogením prostředí
- Spolehlivost komunikace

Základní objekty při používání MPI se nazývají komunikátory. Pomocí nich mohou spolu komunikovat jednotlivé procesy. V rámci jednoho komunikátoru je jednotlivým procesům přiřazen jedinečný identifikátor(rank), díky tomu je možné přesně směrovat komunikaci. Základní komunikátor se jmenuje MPI_COMM_WORLD a při spuštění jsou všechny procesy jeho součásti. Je možné vytvořit si vlastní komunikátory a tím například omezit skupinové posílání dat. Další možností využití vlastních komunikátoru je virtuální topologie, což je přiřazení nových identifikátorů k jednotlivým procesům, tak jak je znázorněno na obrázku 9. Jedná se Kartérzkou strukturu, existuje ještě grafová struktura.



Obrázek 9: Kartérzká struktura

Přenosové funkce

Funkce které slouží pro přenos dat, je možné rozdělit na dvě části a to na:

1. **Point to point komunikaci**. Tato komunikace probíhá pouze mezi dvěma procesy, kde jeden odesílá data (sender) a druhý data přijímá (recevier). Rozlišují se režimy v jakém se zpráva odesílá nebo přijímá:
   - Blokovací
   - Neblokovací
2. **Skupinovou komunikaci**. Zde patří funkce, které pracují s více procesy. Seznam základních skupinových operací jsou popsány v tabulce 1. Existují také variace některých funkcí jako je např. MPI_Allreduce, kde se provede standardní MPI_Reduce, ale poté se výsledek rozešle na ostatní procesy. Je důležité poznamenat, že tato skupina funkci odesílá a přijímá data od všech procesů, které jsou v daném komunikátoru.

MPI_Bcast	Jeden proces odešle stejná všem ostatním procesům
$MPI\_Scatter$	Jeden proces pošle každému procesu jiná data
MPI_Gather	Každý proces pošle data jednomu procesu
MPI_Reduce	Každý proces pošle data jednomu procesu, ale nad těmito daty se provede určitá operace např. sčítání
$MPI\_Barrier$	Jedná se o synchronizaci procesů, kde všechny procesy čekají až se všude spustí tato funkce
$MPI\_Alltoall$	Každý proces pošle data všem ostatním procesům (Velmi náročný na počet přenosů, využívá se minimálně)

Tabulka 1: Skupinové operace

Datové typy, které je možné přenášet jsou téměř totožné se základními datovými typy v jazyce C. Např. MPI_INT, MPI_DOUBLE, MPI_CHAR a další.

Něco málo k historii

První verze MPI standartu byla zhotovena v roce 1994 na které spolupracovalo více jak 40 organizací z USA a Evropy. Protože existovalo několik message-passing systému, tak se snažili využít nejlepších vlastností těchto systémů. O tři roky později byla vydána nová verze MPI-2.0, která přinesla rozšíření v podobě podpory jazyků C++ a Fortran90, dále paralelní I/0, dynamické procesy a jednostrannou komunikaci.Poslední verzí je MPI-2.2, která byla vydána 4.září 2009, ale už se pracuje na MPI-3.

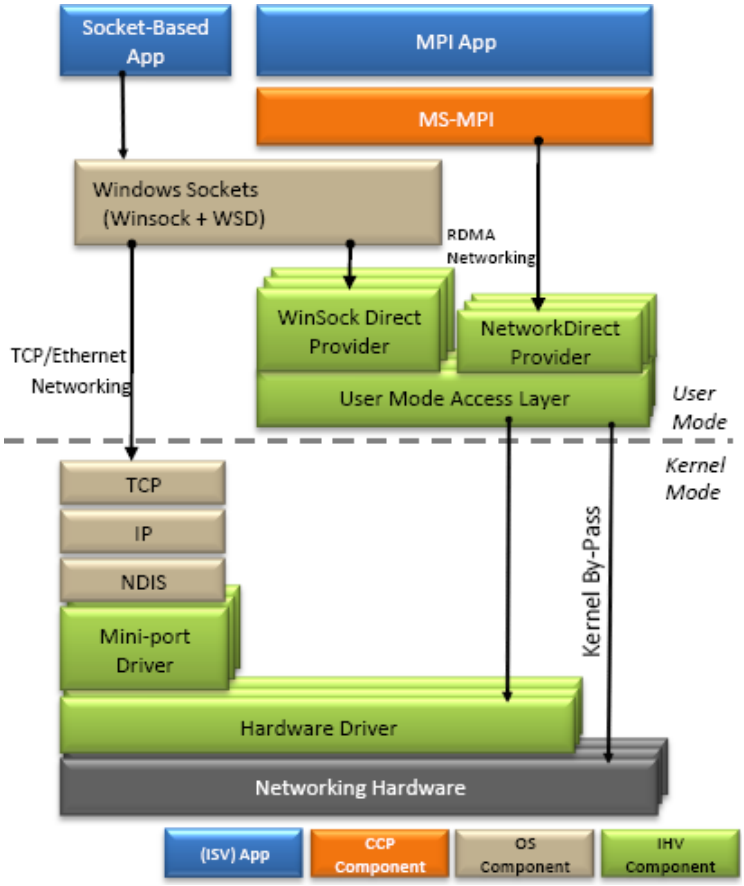
Struktura MPI

MPI se skládá ze dvou částí, první jsou metody, které uživatel může použít a druhou je program, který spouští MPI aplikaci. Jak už bylo dříve napsáno jedná se jen o specifikace. Nejznámější implementace je MPICH2, která je napsána jak pro Windows tak i

Unix. Microsoft napsal vlastní implementaci MS MPI, která byla navržena pro maximální kompatibilitu právě s MPICH2. Ale v některých částech se liší:

- Není kompatibilita se spouštěčem MPI aplikace MPICH2
- Bylo nutné na implementovat zabezpečení pro Windows HPC Server 2008
- Neobsahuje dynamické spuštění procesu a publikování

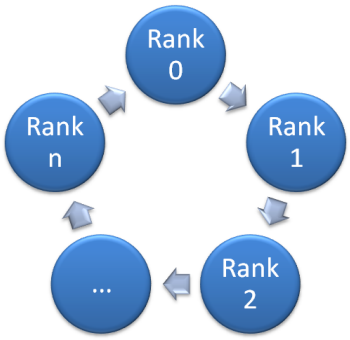
MS MPI využívá Microsoft NetworkDirect protokol pro kompatibilitu s výkonným síťovým zařízením a pro maximální síťový výkon. Ovšem také lze využít jakékoliv síťové připojení, které je podporováno Windows Serverem 2008, jako je např. InfiniBand, 10-Gigabit Ethernet, Myrinet. Na obrázku 10, lze vidět jak NetworkDirect obchází standardní připojení, díky čemuž se zlepšuje výkonost a snižuje zatížení procesoru.



Obrázek 10: Architektura NetworkDirect (převzato z [5], stránka 6)

Zabezpečení se týká přímého napojení na Microsoft Active Directory, kde úlohy spouštěné pod určitým účtem, budou mít stejné oprávnění jako daný účet.

Program pro spouštění MPI aplikací a využívající MS MPI se jmenuje *mpiexec*. Pro MPICH2 se jmenuje *mpirun*.



Obrázek 11: Posílání zpráv v kruhu

Pro vývojáře MPI aplikaci byl do Windows HPC Server 2008 přidán Event Tracing. Jeho předností je, že není nutné jakkoli pozměňovat zdrojový kód, ale pouze stačí při spouštění MPI aplikací přidat parametr *-trace*. Je možné přesně specifikovat, které komunikace chceme zaznamenávat. Protože je celkem mnoho filtrů, které lze použít, doporučujeme se podívat na dokument [4], který mimo jiné popisuje jednotlivé možnosti.

Jak už je výše popsané, MPI standard podporuje jen málo jazyků. Proto vznikají implementace díky kterým je možné psát MPI programy v jazyce C#. Takovýchto implementací je několik a asi nejlepší jsme vybrali MPI.NET. Tato implementace de facto zapouzdřuje MPI funkce C++ a dává je tak k dispozici pro jazyk C# (vytváří wrapper). Při implementace vycházeli z Boos.MPI. Momentálně obsahuje MPI.NET veškeré funkce ze standardu MPI 1.0 a pouze několik z MPI 2.0. Pro vývoj většiny MPI aplikací je to dostatečné, avšak existují výjimky a to např. paralelní zápis procesu do jednoho souboru, měření času. Velkou výhodou je úplná a přehledná dokumentace k jednotlivým funkcím.

3.1 Ukázka MPI v jazyce C++

Pro názornost jsme napsali MPI program, který posílá data v kruhu jak je znázorněno na obrázku 11. Program funguje tak, že rank 0 pošle následujícímu ranku(1) číslo svého ranku. Rank 1 obdrží zprávu, přidá k ní svůj rank a pošle na následující rank. Takhle se to celé opakuje dokud není zpráva na posledním ranku, ten ji nepošle na následující (žádný neexistuje) ale zpět na rank 0, který následně vypíše celou zprávu.

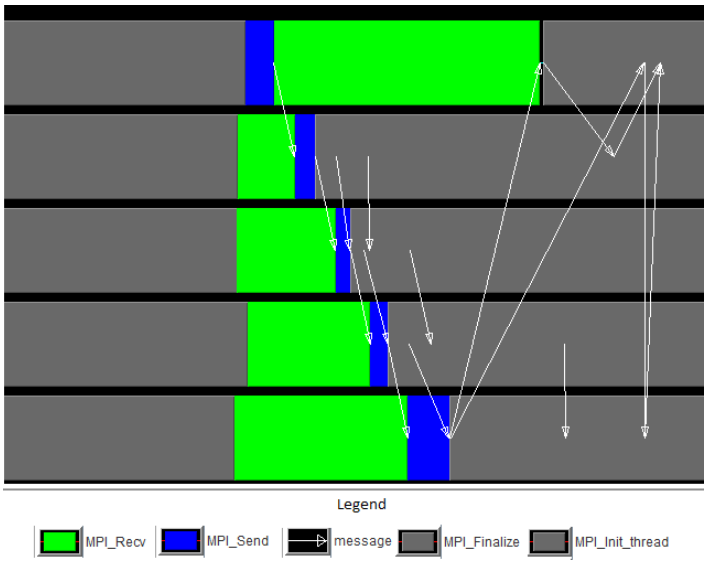
```
int main(int argc, char *argv[])
{
    int numprocs,rank;
    MPI_Status status;
    // Inicilizace MPI
     MPI_Init(&argc,&argv);

    /*
        Zjistím počet procesu na kterých je program spusten a také jejich
         identifikátor - rank
    */
```

```
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

/*
   Testuji zda byl program spusten pouze na jednom procesoru, pokud ano
   vypisu zprávu a ukoncim program, v opacnem pripade posilám data mezi procesory
 */
if (numprocs==1)
{
    cout<<"Pouze jeden procesor"<<endl;
    MPI_Finalize();
    return 0;
}
/*
   Hodnota rank, je jedinecny identifikator procesoru. Cislovan od 0.
   Prvni rank vkruhu musi nejdrive poslat svoji hodnotu a pak cekat na data ktera
   posle posledni rank.
   Ostatni ranky nejdrive cekaji na data, pote pridaji svoji hodnotu a nasledne poslou
   zpravu dale
 */
if (rank==0)
{
    char retezec[500];
    itoa(rank,retezec,10);
    int pocetPrvku=strlen(retezec);
    /*
     Protoze posilam pole znaku o urcite delce, musim nejprve zdelit prijmaci strane
     jak velke pole budu posilat.
     Proto poslu nejdrive pocet prvku v poli a nasledne cele pole. Hodnoty
     posilam na rank 1.
    */
    MPI_Send(&pocetPrvku,1,MPI_INT,rank+1,1,MPI_COMM_WORLD);
    MPI_Send(retezec,pocetPrvku,MPI_CHAR,rank+1,1,MPI_COMM_WORLD);

    /*
     Nevim jak velke pole mam ocekavat, proto obdrzim nejdrive pocet prvku a nasledne
     cele pole. Ocekavam data z ranku (numprocs-1) ->tzn z posledniho ranku
     */
    MPI_Recv(&pocetPrvku,1,MPI_INT,numprocs-1,1,MPI_COMM_WORLD,&status);
    MPI_Recv(retezec,pocetPrvku,MPI_CHAR,numprocs-1,1,MPI_COMM_WORLD,&status);
    retezec[pocetPrvku]='\0';

    cout<<"Seznam ranku:"<<retezec<<endl;
}else
{
    char retezec[500],pomocna[10];
     int pocetPrvku=0;
    MPI_Recv(&pocetPrvku,1,MPI_INT,rank-1,1,MPI_COMM_WORLD,&status);
    MPI_Recv(retezec,pocetPrvku,MPI_CHAR,rank-1,1,MPI_COMM_WORLD,&status);
    retezec[pocetPrvku]='\0';

    itoa(rank,pomocna,10);
    strcat(retezec,",");
    strcat(retezec,pomocna);
```

```
        pocetPrvku=strlen(retezec);

        /*
            Pokud procesor ma rank, ktery je posledni, tak posle zpravu na zacatek kruhu
            tudiz na rank 0. V opacnem pripade posle zpravu nasledujicim ranku
         */
        if (rank==numprocs-1)
        {
            MPI_Send(&pocetPrvku,1,MPI_INT,0,1,MPI_COMM_WORLD);
            MPI_Send(retezec,pocetPrvku,MPI_CHAR,0,1,MPI_COMM_WORLD);
        }else
        {
            MPI_Send(&pocetPrvku,1,MPI_INT,rank+1,1,MPI_COMM_WORLD);
            MPI_Send(retezec,pocetPrvku,MPI_CHAR,rank+1,1,MPI_COMM_WORLD);
        }

    }
    // Ukonceni prace s MPI
    MPI_Finalize();
    return 0;
}
```

Výpis 2: Ukázkový program v C++

Na obrázku 12 je vidět trasování programu, který je spuštěný na 5 procesorech. Zobrazení je pořízeno z programu *Jumpshot*. Každý řádek představuje jeden procesor v čase, kde nalevo je nejmenší čas a napravo největší. Modrou barvou je vidět jak dlouho jednotlivé procesory posílaly data, zelenou jak dlouho čekaly na data a bílé šipky znamenají směr přenosu zprávy. I z toho výstupu je poznat že program se chová podle zadání.



Obrázek 12: Volání MPI funkci v jazyce C++

3.2 Ukázka MPI v jazyce C#

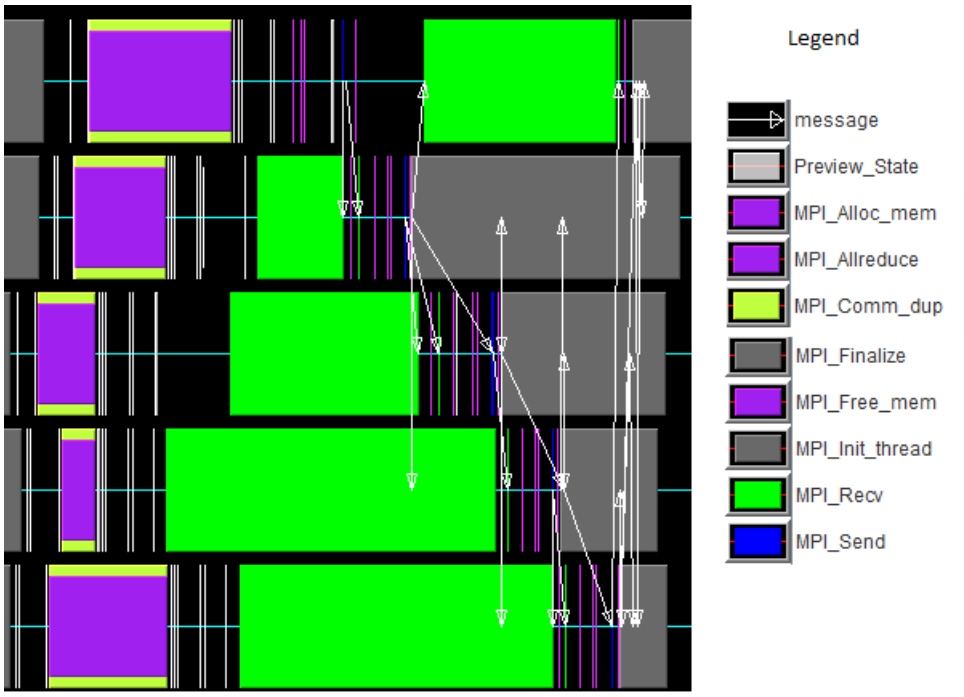
Tento program má totožné zadaní jako předcházející ukázka v jazyce C++. Kód je jednoduší, kratší a přehlednější (není potřeba nastavovat tolik parametru).

```
class Program
{
    static void Main(string[] args)
    {
        using (new MPI.Environment(ref args))
        {
            //Ziskani komunikatoru, pro veskerou dalsi praci
            Intracommunicator comm = Communicator.world;

            /*
             * Testuji zda byl program spusten pouze na jednom procesoru, pokud ano
             * vypisu zpravu a ukoncim program, v opacnem pripade posilam data mezi
                 procesory
             */
            if (comm.Size == 1)
            {
                Console.WriteLine("Pouze jeden procesor");
                return;
            }

            /*
             * Hodnota Rank, je jedinecny identifikator procesoru. Cislovan od 0.
             * Prvni rank vkruhu musi nejdrive poslat svoji hodnotu a pak cekat na data ktera
             * posle posledni rank.
             * Ostatni ranky nejdrive cekaji na data, pote pridaji svoji hodnotu a nasledne
                 poslou
             * zpravu dale
             */
            if (comm.Rank == 0)
            {
                string retezec=Convert.ToString(comm.Rank);
                comm.Send(retezec, comm.Rank + 1, 1);              //Poslu retezec na rank
                    cislo 1

                /*
                 * Hodnota comm.Size, obsahuje pocet procesoru na kterych byl program
                     spusten.
                 * Program ceka na zpravu od posledniho ranku, tudiz z hodnotou comm.Size
                     -1
                 */
                comm.Receive(comm.Size - 1, 1, out retezec);

                Console.WriteLine("Seznam ranku:" + retezec);
            }
            else
            {
                string buffer;
                comm.Receive(comm.Rank - 1, 1, out buffer);        //Obdrzeni zpravy od
                    predchoziho ranku
```

```
                buffer = buffer +","+ Convert.ToString(comm.Rank);

                /*
                 * Pokud procesor ma rank, ktery je posledni, tak posle zpravu na zacatek
                     kruhu
                 * tudiz na rank 0. V opacnem pripade posle zpravu nasledujicim ranku
                 */
                if (comm.Rank==comm.Size-1)
                    comm.Send(buffer, 0, 1);
                else
                    comm.Send(buffer, comm.Rank + 1, 1);
            }
        }
    }
}
```

Výpis 3: Ukázkový program v C#

Opět na obrázku 13 je vidět trasování programu. Sice samotný kód má menší počet řádků a je čitelnější, ale komunikace je mnohem náročnější (alespoň při počáteční inicializaci) a skrývá volání MPI funkcí, které programátor v programu vůbec nevolal např. MPI_Reduce.



Obrázek 13: Volání MPI funkci v jazyce C#

3.3 Testy a porovnání

V této kapitole bychom chtěli porovnat jak velký časový rozdíl je mezi přenosem zpráv psaných v jazycích C++ a C#, při použití MPI funkcí (předem jsem předpokládal, že jazyk C++ bude rychlejší) a porovnat rychlost aplikace spuštěné na Windows HPC Serveru 2008 a uzlu Quad. Výsledné časy popisují čistě dobu běhu algoritmů, bez jakékoli inicializace nebo výpisu.

Při spouštění programu jsme využívali školní HPC server, který má jeden hlavní uzel a tři výpočetní uzly. Každý z uzlu má 8 procesorů a 16 GB paměti, hlavní uzel má pouze 4 procesory a 4 GB paměti. Propojení mezi jednotlivými zařízeními je pomocí jediného síťového rozhraní.

Je nutné si uvědomit, pokud je aplikace spuštěna např. na 8 procesorech, tak není žádná komunikace po sítí, protože se jedná o komunikaci v rámci samotného uzlu. Proto čas výpočtu na 9 procesorech může být (např. při velké komunikaci) větší než při spuštění na 8 procesorech.

C++ versus C#

Principem testů je zjistit čas potřebný k přenesení zpráv stejné velikosti a typu mezi procesory. Aby byla komunikace po síti, tak jsme spouštěli výpočet na dvou výpočetních uzlech tzn. 16 procesorech. Napsali jsme dva programy, jeden v jazyce C++ druhý v jazyce C#, v obou programech jsme nad stejnými daty volal stejné funkce a výsledné časy lze vidět v tabulce 2. V tabulce není funkce *Scatter*, protože její fungování v MPI.NET se trochu liší od standardní implementace MPI.

Při testování jsme zjistili, že pokud tu samou funkci zavolám dvakrát za sebou tak čas druhé funkce bude podstatně kratší než čas funkce první.

Název	Čas u C++(ms)	Čas u C#(ms)
Point to Point(posíláno v rámci jednoho uzlu)	0.236	0.3290
Point to Point	0.869	9.7220
Gather	0.458	7.9558
Broadcast	4.314	14.8570
Reduce	1.614	17.3023

Tabulka 2: Porovnání časů u MPI funkci

Doba přenosu třídy pomoci MPI.NET

V tomto testu nás zajímalo, že když si vytvoříme vlastní třídu a budeme jí přenášet z jednoho procesoru na druhý, tak o kolik více času bude potřeba než kdybych proměnné tohoto našeho typu přenášeli odděleně. Náše nadefinovaná třída obsahuje pouze dvě proměnné, jednu typu int a druhou typu double. Vytvořili jsme si pole této třídy o velikosti 100 a dále ještě dvě pole taktéž o velikosti 100, ale jedno bylo typu int a druhé typu double. Získali jsme dvě časové hodnoty, jednu posláním pole naší třídy a druhou, když jsme posílali ta další dvě pole, ale neposílali jsme je v celku ale po hodnotách int a

double na střídačku. Výsledné časy jsou vidět v tabulce 3, ale rozdíly jsou opravdu velké, proto pokud není nezbytně nutné posílat vlastní typ tak se tomuto způsobu přenosu raději vyhnout.

Typy proměnných	Čas potřebný k přenosu dat (ms)
Vlastní třída	26.4949
Oddělené proměnné(int a double)	5.7338

Tabulka 3: Porovnání přenosu vlastního typu a oddělených proměnných

Windows HPC Server 2008 versus Quad

Paralelizovali jsme známý Floyduv-Warshalluv algoritmus pro nalezení všech nejkratších cest mezi jednotlivými dvojicemi vrcholů ohodnoceného orientovaného grafu, reprezentovaného svou matici sousledností (vzdáleností). Školní systém Quad má pouze 4 procesory a 4 GB sdílené operační paměti. Tento program jsme napsali v jazyce C++, abychom jej mohli bez jakýkoliv úprav spustit na obou systémech a výsledky lze vidět v tabulkách 4 a 5. Je nutné vzít na zřetel, že na Quadu se sice spustil program na 10 procesech, ale ve skutečnosti běžel pouze na 4 procesorech. Kdežto na Windows HPC běžela aplikace na 10 procesorech. Celý zdrojový kód je na přiloženém CD.

Počet vrcholů	Čas sekvenčně	Čas pro 4 procesy	Čas pro 10 procesů
5	0.000001s	0.000242s	0.040186s
100	0.018654s	0.005986s	0.057055s
250	0.285945s	0.076062s	0.155998s
500	2.292847s	0.579609s	1.114617s

Tabulka 4: Výsledky testů na Quad

Počet vrcholů	Čas sekvenčně	Čas pro 4 procesy	Čas pro 10 procesů
5	0.000003s	0.001424s	0.005464s
100	0.010098s	0.005055s	0.012172s
250	0.148975s	0.050420s	0.041783s
500	1.166520s	0.342693s	0.191889s

Tabulka 5: Výsledky testů na Windows HPC Server 2008

4 Neuronové sítě

Neuronová síť, jak ji popsal profesor Vondrák[7], má několik vlastností:

- Umělá neuronová síť je vytvořena na základě biologické umělé sítě, tudíž její chování by mělo být stejné nebo alespoň podobné.
- Práce s informacemi probíhají v rámci celé neuronové sítě.
- Při učení se mění hlavně vazby mezi jednotlivými neurony, kde ty které vedou ke správné odpovědi jsou posilovány a ty co naopak vedou ke špatné odpovědi jsou zeslabovány.
- Základní vlastností je učení. Kde záleží na trénovací množině, díky které se naučí neuronová síť jak převádět daný vstup na správný výstup. Nejvíce náročnou fází neuronové sítě je právě její učení, která může být i zdlouhavá. Kdežto samotný průchod neuronovou sítí už je rychlý.

Neuronová síť se skládá z jednotlivých neuronů, kde nejpoužívanější model se nazývá perceptron. Základní perceptron je definován pomoci vztahu :

$$y = F(\sum_{i=1}^{n}(w_i x_i) - \theta) \tag{1}$$

Kde *y* je výstup, w_i je váha mezi neurony, x_i je výstup z předešlého neuronu, θ je prahová hodnota, **F** je aktivační funkce neuronu a *n* je počet neuronů. Pokud by tento součet byl větší než určitá prahová hodnota(θ), poté bude na výstupu hodnota 1 v opačném případě hodnota 0.

Tento vztah však můžeme upravit pokud zavedeme substituci takovou, že za w_0 dosadíme $-\theta$ a za x_0 dosadíme 1. Poté dostaneme vztah následující:

$$y = F(\sum_{i=0}^{n}(w_i x_i)) \tag{2}$$

Dalším typem perceptronu je spojitý perceptron, jehož aktivační funkce má tvar sigmoidy, jak je zobrazeno na obrázku 14. Tuto funkci můžeme vyjádřit následovně:

$$F(z) = \frac{1}{1 + e^{-z}} \tag{3}$$

Kde *z* se rovná $\sum_{i=0}^{n} w_i x_i$.

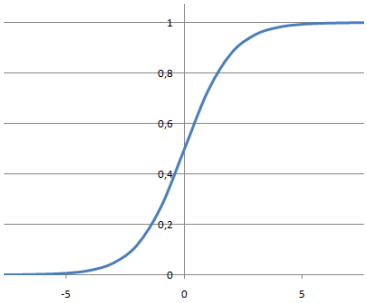
Existuje ještě modifikace této metody, při které se nebudou měnit pouze vazby mezi neurony, ale i samotné neurony. Princip spočívá v tom, že se bude měnit u aktivační funkce navíc ještě strmost a práh.

Vzorec pro takovouto funkci by byl následující:

$$F(z) = \frac{1}{1 + e^{-\lambda(z-\vartheta)}} \tag{4}$$

Kde λ určuje strmost a ϑ je prahová hodnota.

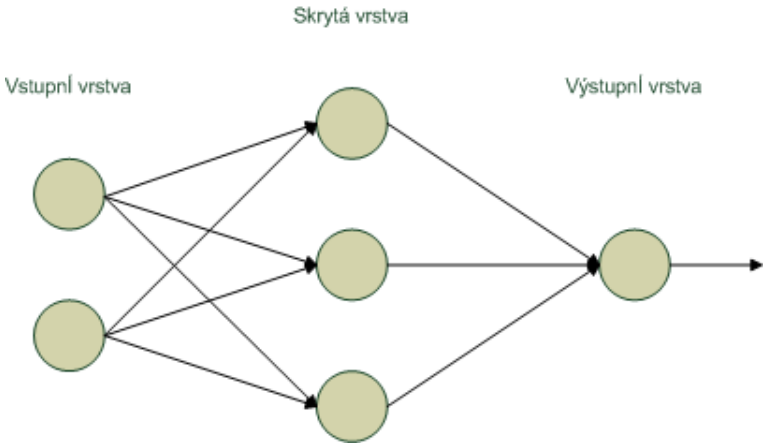
Tuto variantu jsme však při implementaci nepoužili.



Obrázek 14: Průběh aktivační funkce

4.1 Backpropagation

Jedná se o více vrstvou neuronovou síť, která má tři vrstvy jak je vidět na obrázku 15. Jednotlivé vrstvy jsou vstupní, skrytá (zde je možné mít ve skutečnosti více vrstev) a výstupní. Mezi jednotlivými vrstvami jsou neurony úplně propojené, tzn. každý neuron z jedné vrstvy je propojen s každým neuronem vrstvy vyšší.



Obrázek 15: Ukázka neuronové sítě

Jak už je patrné z názvu neuronové sítě, bude se jednat o zpětnou propagaci chyb. Jedná se o síť s učitelem, tzn. je nutné mít trénovací množinu jako pár - vstup a jaký mu přísluší výstup. Tuto metodu je možné rozdělit na tyto tři fáze (jednotlivé vztahy převzány z [7, 6]):

1. Ze vstupů získat výstup neboli dopředné šíření. Nastaví se vstupní vrstva tak, aby jednotlivé neurony dávali výstup podle trénovací množiny (rozsah od 0 do 1). Tyto výstupy jsou dále přeneseny do vyšší vrstvy a upraveny podle jednotlivých synapsí. V této vyšší vrstvě každý neuron vypočítá sumu jemu patřící upravených výstupů z nižší vrstvy a tento výsledek vloží do aktivační funkce. Aktivační funkce dá výsledek v rozsahu 0 až 1(jedná se o spojitý perceptron), což bude i výstup z

neuronu. Tento postup se bude opakovat až do výstupní vrstvy, která dá výstup z neuronové sítě.

2. Vypočítat chyby a ty následně šířit z výstupu zpět na vstup.

   Výpočet změny vah (Δw_i) vypočítáme podle vzorce:

   $$\Delta w_i = -\eta x_i \delta + \mu(\Delta w_i') \tag{5}$$

   Kde je η učící poměr, x_i je výstup z předešlého neuronu, δ vypočtená chyba, μ je stupeň závislosti na předchozí změně a $\Delta w_i'$ je předchozí změna vah.

   Výpočet chyby se liší podle vrstvy, tudíž je nutné rozlišit jestli se jedná o výpočet chyby u výstupní a nebo u skryté vrstvy.

   Pro výstupní vrstvu se chyba vypočítá skutečný výstup mínus očekávaný výstup. Tento výpočet se provede na všech neuronech výstupní vrstvy. Vzorec je následující:

   $$\delta_k = (y_k - o_k)F(z)(1 - F(z)) \tag{6}$$

   Kde o_k je očekávaný výstup, y_k je skutečný výstup a *F(z)* je aktivační funkce $\frac{1}{1+e^{-z}}$.

   Pro skrytou vrstvu, je výpočet složitější, protože se musí vynásobit chyba neuronu z vyšší vrstvy a váha (synapse), která spojuje dané dva neurony. Tento výpočet se provede nad všemi neurony vyšší vrstvy, protože se jedná o úplně propojeni. Tento výpočet se provede na všech neuronech dané vrstvy. Vzorec pro skrytou vrstvu je následující:

   $$\delta_j = (\sum_{i=1}^{m} w_i \delta_i)F(z)(1 - F(z)) \tag{7}$$

   Kde m je počet neuronů ve vyšší vrstvě, δ_i je chyba í-tého neuronu z vyšší vrstvy, w_i je váha mezi aktualním neuronem a í-tým neuronem vyšší vrstvy.

3. Aplikovat změny vah podle vypočítaných hodnot.

Tento postup se bude provádět dokud se nevyčerpá trénovací množina a poté se spočítá celková chyba. Pokud je tato chyba větší než přednastavená hodnota, tak se celý proces zopakuje.

Výpočet celkové chyby (E) je následující:

$$E = \frac{1}{2}\sum_{i=1}^{p}\sum_{j=1}^{l}(y_j - o_j)_i^2 \tag{8}$$

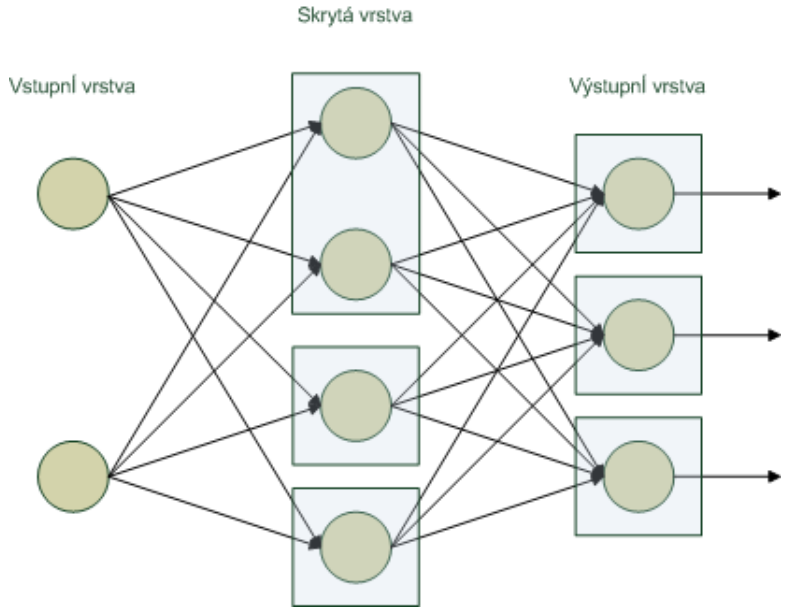
Kde y_j je skutečný výstup, o_j je očekávaný výstup, l je počet vzorů ve trénovací množině a p je počet neuronů ve výstupní vrstvě.

4.1.1 Implementace

Sekvenční řešení jsme na implementovali pomocí skript[7, 6]. Jako první bylo třeba určit jaký proces bude počítat jaké neurony. Protože najednou se počítají neurony pouze v jediné vrstvě a ostatní vrstvy čekají na data, proto jsme každou vrstvu rozdělili na jednotlivé bloky a každému bloku jsme přiřadili proces, jak je vidět na obrázku 16. Pouze vstupní vrstva není rozdělená, protože všechny procesy musí znát vstupy. Toto blokové rozdělení je z určitého důvodu, když přenášíme data z jedné vrstvy do druhé tak můžeme použít MPI funkci GatherFlatened, pomocí které se data přenesou v určitém pořadí. MPI.NET obsahuje ještě variantu této funkce AllGatherFlatened, která by měla přenést výsledek na všechny procesory, bohužel tato funkce nepracuje tak jak by měla. Z tohoto důvodu jsme výsledek museli nechat přenést na jeden z procesů (všechny podobné přenosy provádíme přes proces s rankem 0) a následně MPI funkci Broadcast přenést na ostatní procesy.

Po vytvoření topologie neuronové sítě a rozdělení do bloků, jsme řešili práci s pamětí při přenášení dat mezi jednotlivými vrstvami. V MPI.NET jsou MPI funkce, které přijímají data, navrženy tak, že se pro data alokuje nová paměť nebo se použije už alokovaná paměť. Protože přenosů je velké množství alokovali jsme paměť na začátku programu, abychom zabránili neustálé alokaci/dealokaci paměti.

Díky skutečnosti že posíláme chyby(vytvářím vlastně jejich kopii) je možné sloučit body 2 a 3, tzn. po výpočtu chyby a odeslání můžeme hnedka aktualizovat váhy. Výhodu jakou nám tato operaci přinese je zrychlení, protože nemusíme znova procházet celou síť a aktualizovat váhy.



Obrázek 16: Ukázka rozdělení vrstev na procesy

Výstup

Jako výstup jsme se rozhodli použít standardní výstup (při spuštění na Windows HPC Serveru 2008 je přesměrován do souboru). Je to jeden ze způsobů, jak můžeme paralelně ukládat data do souboru. Nevýhodou tohoto zápisu je, že se jedná o textový soubor.

4.1.2 Experimenty

Časy které jsou uvedeny v tabulkách 6 a 7, jsou z spočítány tak že celkový čas potřebný k naučení neuronové sítě jsme vydělili počtem epoch, které bylo nutné projít. K tomuto výpočtu jsme se rozhodli na základě toho, že neurony na počátku jsou nastaveny na náhodnou malou hodnotu a celkový potřebný čas k výpočtu se bude lišit každým novým spuštěním. Existují časové rozdíly jestli musí spolu procesy komunikovat přes síť a nebo pouze v rámci jednoho uzlu (tyto jsou označeny hvězdičkou). Slovem epocha máme namysli, vyčerpání všech vzorů z trénovací množiny.

Testovali jsme výkonnost algoritmu dvěma následujícími způsoby:

- První jsme zabývali neuronovou sítí, která měla jen tři vrstvy (vstupní, skrytou a výstupní) a měnili se pouze počet neuronů ve skryté vrstvě. Výsledky toho testu jsou zobrazeny v tabulce 6.

Počet procesorů	**Časy potřebné k výpočtu jedné epochy, v sekundách**		
	Vrstvy: 64-400-16	**Vrstvy:64-500-16**	**Vrstvy:64-600-16**
1*	0.9404	1.1665	1.4113
4*	0.4041	0.4699	0.5629
7*	0.3540	0.41238	0.4786
10	0.5035	0.6090	0.65047
13	0.5151	0.6069	0.7020
19	0.6404	0.7065	0.8244

Tabulka 6: Závislost počtu procesorů na počtu neuronů v jediné skryté vrstvě

- Druhý test se zabýval zvyšováním počtu skrytých vrstev. Výsledky tohoto druhého testu jsou zobrazeny v tabulce 7.

4.2 Kohonenova mapa

Tento druh sítě se podstatně liší od backpropagation. Jedná se o síť bez učitele, tudíž stačí pouze na vstup přivést trénovací data a síť si sama určí rozvržení (vytvoří shluky). Tato síť má pouze dvě vrstvy, kde spodní vrstva je vstupní, která je propojena se všemi neurony vrchní vrstvy, výstupní. Taktéž jsou propojeny mezi sebou neurony ve výstupní vrstvě. Toto dvourozměrné zobrazení se používá nejčastěji. Principem této sítě je převedení složitější datové struktury na jednodušší zobrazení a to tak, že pokud jsou si vstupní data podobná, tak výstupní neurony, které je mají charakterizovat, budou blízko u sebe.

Počet procesorů	Časy potřebné k výpočtu jedné epochy, v sekundách	
	Vrstvy:64-200-100-16	Vrstvy:64-200-100-100-16
1*	1.3365	1.8663
4*	1.0559	1.6819
7*	1.0634	1.7548
10	1.6055	2.4554
13	1.7784	2.7407
19	2.0294	3.1545

Tabulka 7: Závislost počtu procesorů na počtu skrytých vrstev

4.2.1 Implementace

Pro implementaci Kohonenovy mapy jsme vycházeli se sekvenčního řešení. Jednotlivé kroky, které je nutné provést u sekvenčního řešení jsou popsány v tabulce 8. Sekvenční řešení jsme na implementovali pomocí skript[7, 6] a internetového seriálu[9].

1.		Vytvořit síť neuronu, přičemž se nastaví váhy jednotlivých neuronu na malé hodnoty.
2.		Nastavit počáteční míru sousedství, rychlost učení a počet iteraci.
3.	(a)	Načíst vstupní data.
	(b)	Vypočítat u všech neuronu vzdálenost podlé vstupních dat.
	(c)	Najít nejmenší vzdálenost mezi neurony.
	(d)	Upravit neuron s nejmenší vzdáleností a jeho přilehlé okolí.
4.		Opakovat bod 3 dokud není splněn počet iteraci.

Tabulka 8: Postup implementace

Při paralelizaci jsme postupně procházeli jednotlivé body a hledali řešení, které bude co nejoptimálnější (Co nejmenší počet přenosu mezi procesory, neprocházet celou síť ale jen část apod.).

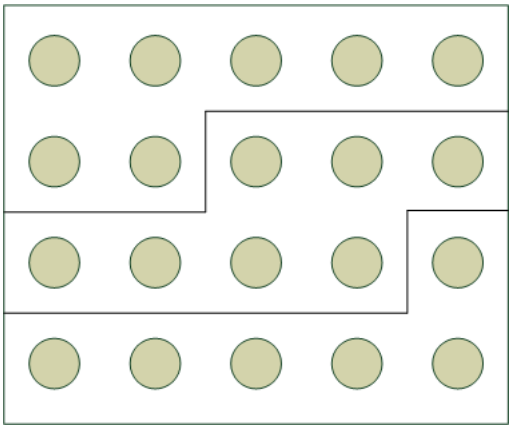
Tabulka 8, bod 1

Prvotní nápad byl rozdělit dvourozměrnou neuronovou síť na čtvercové (obdélníkové) bloky, ale narazili jsme na překážku a to v tom, že nedokážeme jakoukoli neuronovou síť o velikosti M x N rozdělit na přibližně stejně velké bloky při různém počtu procesorů. Proto jsme neuronovou síť rozdělili na bloky, kde jednotlivé neurony jdou v pořadí po sobě. Počet neuronu v takových to blocích spočítám pomocí vzorce:

$$PocetNeuronu = \frac{m}{n} \tag{9}$$

Kde m je celkový počet neuronu a n je celkový počet procesů.

Protože výsledek nemusí být celé číslo, tak postupně přidáváme k jednotlivým blokům neuron navíc, tak aby jejich celkový součet odpovídal hodnotě m. Toto přidávání provádíme postupně od bloku 1 až po n−1 bloků. Výhodou toho uspořádání je, že počet neuronu v jednotlivých blocích se může maximálně lišit o jeden neuron. Znázorněno na obrázku 17. Na tomto obrázku vidíme mapu neuronové sítě o velikosti 5 x 4, kterou chceme spustit na 3 procesorech (proto rozdělení na 3 části). Důležitá věc je, že každý procesor má v paměti jen ty prvky, které mu náleží a žádné jiné. Nikde neexistuje kompletní mapa (s výjimkou nakonec uložených dat do souboru), ale jen její části. Každý procesor má jednorozměrné pole o velikosti počtu jeho neuronu a každý neuron v sobě má pozici v mapě.



Obrázek 17: Rozdělení neuronové sítě na bloky

Tabulka 8, bod 2

Počáteční míru sousedství jsme na počátku nastavili na jednu polovinu kratšího rozměru sítě. Rychlost učení je nastavena na 0.5 a počet intervalu se mění podle zadání.

Tabulka 8, bod 3a

Nejprve načítáme všechna vstupní data ze souboru do paměti. Při každé iteraci vložíme následující data na vstup. Pokud jsou všechna vstupní data použita a počet iterací nebyl splněn, začneme na vstup opět dávat načtená data od začátku.

Tabulka 8, bod 3b

Zde jsme naplno využili paralelismus. Každý proces musí na každém jemu přiřazeném neuronu spočítat vzdálenost a zároveň hledat její nejmenší hodnotu. Vzdálenost se spočítá pomocí vzorce:

$$vzdalenost = \sqrt{\sum_{i=1}^{n}(x_i(t) - w_i(t))^2} \tag{10}$$

Kde n je počet vstupů, x_i je vstup a w_i je váha mezi vstupem a neuronem.

Protože máme neurony přibližně stejně rozdělené (podle bodu 1), tak můžeme říct že spočítaní vzdálenosti všech neuronu bude trvat nejvýše tak dlouho, jak dlouho trvá spočítat první blok neuronů.

Tabulka 8, bod 3c

Jedná se o jedinou komunikaci mezi procesy. Každý proces má jednu nejmenší vzdálenost a tu musí porovnat s ostatními procesy. Sice se porovnává jediná hodnota, ale potřebujeme ještě vědět pozici neuronu s touto hodnotou. Proto jsme si vytvořili vlastní proměnnou, která obě tyto informace obsahuje. Při porovnání jsme využili MPI funkci Allreduce. Tato funkce s parametrem MPI_MIN (tato operace už je předdefinována, ale je možné si vytvořit vlastní operace) všechna data z procesorů porovná mezi sebou a vrátí nejmenší hodnotu. Provádějí se ve skutečností tři operace: Stažení dat na jeden procesor, nalezení minima a následné odeslání výsledku všem zúčastněným. Po provedení této funkce mají všichni minimální hodnotu i její pozici.

Tabulka 8, bod 3d

Velikost oblasti, která se bude měnit určíme pomocí vzdálenosti od vítězného neuronu. Tato vzdálenost se spočítá pomocí vzorce:

$$\mu(t) = \mu_0 \exp(-\frac{t}{\lambda}) \tag{11}$$

Kde μ_0 je počáteční míra sousedství a λ je časová konstanta.

Časová konstanta se vypočítápomocí vzorce :

$$\lambda = \frac{k}{\log(\mu_0)} \tag{12}$$

Kde k je celkový počet iterací a μ_0 je počáteční míra sousedství.

Všechny neurony v této oblasti budou aktualizovány pomocí vzorce:

$$w_i(t+1) = w_i(t) + \eta(t)\delta(t)(x_i(t) - w_i(t)) \tag{13}$$

Kde η je rychlost učení, w_i je je váha mezi vstupem a neuronem, x_i je vstup a δ určuje vzdálenost aktuálního neuronu k vítěznému neuronu, čím je vzdálenost větší, tím menší změna se provede.

Rychlost učení se postupně snižuje a to podle vzorce:

$$\eta(t) = \eta_0 \exp(-\frac{t}{\lambda}) \tag{14}$$

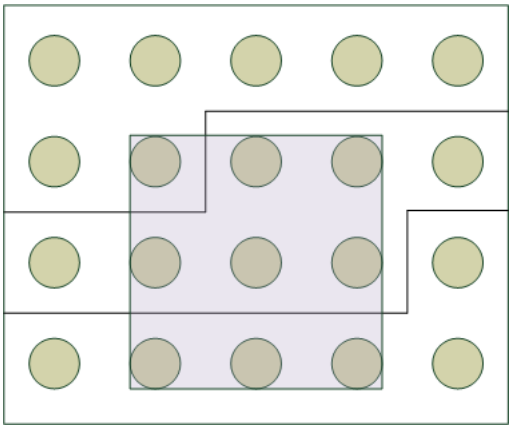
Kde η_0 je počáteční rychlost učení a λ je časová konstanta.

Hodnota δ se vypočítá následovně:

$$\delta(t) = \exp(-\frac{r^2}{2\mu^2(t)}) \tag{15}$$

Kde r je Euklidovská vzdálenost mezi aktuálním neuronem a vítězným neuronem.

Při aktualizaci vah neuronu není nutné procházet celou mapu, ale stačí jen čtverec, který má délku hrany rovnou dvakrát poloměr kružnice se středem na pozici vítězného neuronu. Díky tomuto čtverci jsme museli mít uspořádané neurony do bloků. V sekvenčním programu není problém tento čtverec projit. Ovšem u paralelního tu existují přechody mezi procesory a je nutné dávat si na to pozor. Jak je vidět na obrázku 18, tento čtverec zasahuje přes všechny 3 procesory a na každém přes jiný počet neuronů. Naše řešení si na každém procesu vypočítá daný čtverec (jeho levý horní roh a pravý spodní roh) a následně jej upraví podle bloku neuronů, který mají. Byl tu problém, jak už jsme psali v bodě 1, každý proces má jednorozměrné pole a za sebou naskládané neurony, tudíž nemůžeme přímo adresovat neurony v mapě, ale musíme je ještě přepočítat na jednorozměrné pole. Nejhorší možná délka výpočtu bude záviset na procesu, který počítá nejvíce neuronu ve čtverci. Může nastat případ, že daný čtverec bude obsahovat pouze neurony z jednoho procesu, pak v této části výpočtu nebude žádné zrychlení, protože čas bude stejný jako při sekvenčním řešní.



Obrázek 18: Zobrazení výbraných neuronů

Výstup

Jako výstup jsme se rozhodli používat binární soubor. Protože neexistuje celková mapa neuronové sítě, tak jsme se původně rozhodli využít funkce MPI2 pro práci se soubory. Existují funkce, které umožňují aby více procesorů mohlo zapisovat do jediného souboru. Bohužel implementace MPI.NET tyto funkce neobsahuje a tudíž jsme museli data ze všech procesoru postupně přenášet na procesor s rankem 0 a ten zapisoval do souboru. Pokaždé když jsme přenesl data z dalšího procesoru tak jsme přepsali původní, jinak by mohl nastat případ, že se data nevlezou do paměti. Ovšem i toto řešení se při testování ukázalo jako nepraktické, pro malý počet neuronu v neuronové síti je to pořádku, ale pro větší sítě samotné ukládání trvá několikanásobně déle než samotné učení. Pro jsme do implementoval možnost ukládání dat po jednotlivých procesech do samostatných souborů.

4.2.2 Experimenty

Protože testujeme aplikace na školním Windows HPC Serveru 2008, může nastat případ takový, že aplikace pokud byla spuštěna na jednom uzlu, tak nemusí vůbec komunikovat přes síť (jinak řečeno může být rychlejší). Takové případy jsou v tabulkách označeny hvězdičkou. Počet opakování je nastaven na 10 000.

Prováděli jsme dvě následující měření:

- Počet vstupů pro všechny testy jsme nastavili na hodnotu tři a měnili jsme pouze velikost neuronové sítě, jak je zobrazeno v tabulce 9.

Počet procesorů	**Časy potřebné k výpočtu, ve tvaru hh:mm:ss**		
	Rozměr: 250x250	**Rozměr: 500x500**	**Rozměr: 1000x1000**
1*	00:02:01	00:08:47	00:36:33
4*	00:00:34	00:03:12	00:13:36
7	00:00:20	00:02:00	00:08:04
10	00:00:18	00:01:13	00:05:32
13	00:00:15	00:00:53	00:04:30
16	00:00:14	00:00:38	00:03:39
19	00:00:15	00:00:35	00:03:09

Tabulka 9: Závislost počtu procesorů na velikosti sítě

- V tomto testu jsme naopak měli napevno nastavený rozměr neuronové sítě na 500x500 a měnili jsme pouze počet vstupů. Výsledky měření jsou zobrazeny v tabulce 10.

Počet procesorů	**Časy potřebné k výpočtu, ve tvaru hh:mm:ss**		
	Pro 3 vstupy	**Pro 6 vstupů**	**Pro 9 vstupů**
1*	00:08:47	00:13:11	00:18:55
4*	00:03:12	00:03:48	00:05:33
7	00:02:00	00:02:16	00:03:02
10	00:01:13	00:01:34	00:02:12
13	00:00:53	00:01:14	00:01:50
16	00:00:38	00:00:57	00:01:29
19	00:00:35	00:00:50	00:01:14

Tabulka 10: Závislost počtu procesorů na počtu vstupů

5 Distribuované datové struktury

Cílem tohoto projektu je vytvořit knihovny, které se zabývají paralelními výpočty. Tyto knihovny poskytnou silný nástroj pro výpočet náročných matematických úloh (ať už se jedná o časovou a nebo paměťovou náročnost). Vychází se z už hotové sekvenční verze OOSol (Object Oriented SOLvers), která byla vyvíjená na katedře aplikované matematiky[24]. Nynější verze se také vyvíjí na katedře aplikované matematiky a vedoucím vývojového týmu je docent Vít Vondrák (kontakt: vit.vondrak@vsb.cz). Jádro programu je napsáno v jazyce C++, pro distribuce dat mezi procesy je primárně použito MPI, ale je možné pomocí wrapperu i jinou distribuci (např. WCF). Díky tomu, že se používá jazyk C++ a specifikace MPI2, je možné vyvíjet a testovat knihovny ať už na OS Linux (zpočátku se preferovala pouze tato možnost) a nebo na Windows HPC.

V rámci OOSolu je našim úkolem zajišťovat posílání datových struktur pomocí MPI a dále testovat a opravovat nově napsané metody. Proto zde uvedeme některé vlastnosti, které jsme vyvíjeli.

Při práci s velkou matici, není potřeba aby ji všechny procesy znali celou, ale stačí pouze určité části (bloky). Proto se podle předem daného popisu rozdělí matice na malé bloky(nemusí být stejné) a ty jsou dále distribuovány na jednotlivé procesy. Zde je několik problémů, které bylo nutné vyřešit. Dle zadání, může jeden proces zpracovávat pouze jeden blok matice a proces s rankem 0 je řídící (např. určuje kdo má co počítat a pod.). Pro lepší pochopení se podívejte na příklad 5.1.

Příklad 5.1
Úloha se spustí na 13 procesorech (tzn. ranky mají hodnotu od 0 do 12). Uživatel zadá velkou matici a její rozdělení např. na 6 bloků. Proces s rankem 0 přidělí jednotlivé bloky rankům 1 až 6. Uživatel zadá druhou matici a její rozdělení na 4 bloky, tak opět řídící proces přidělí tyto bloky a to rankům 7 až 10. Nyní zůstanou pouze 2 bloky nevyužité. Pro další použití je možné přidělit pouze dva bloky a nebo uvolnit už přidělené bloky. Pokud uvolní uživatel první matici z procesorů, tak dostane možnost přidělit nové bloky rankům 1 až 6, 11 a 12. ■

Dalo by se říct, že se jedná o podobný problém jako je fragmentace paměti. Řešení tohoto problému je následující, řídící proces si vytvoří jednorozměrné pole, které odpovídá jednotlivým procesům a nastaví jednotlivé hodnoty na -1. Když přijde požadavek na přidělení jednotlivých bloků, prohledá toto pole zleva do prava a ty položky, které obsahují -1 (jsou volné procesy) přidělí jim dané bloky a změní hodnotu v poli. Protože všechny procesy čekají jestli jim není náhodou přidělen nějaký blok, tak je nutné posílat všem zprávu, který rank má přidělený jaký blok. Zde se provedlo ještě jedno urychlení spočívající v tom, že ranky které pracují na stejné matici mají vytvořený nový komunikátor, pomoci kterého posílají veškeré skupinové funkce (např. MPI_Bcast se nebude posílat všem procesům, ale pouze těm co pracují na stejné matici). I když je vytvořen nový komunikátor, ten původní je pořád platný. Platnost nového komunikátoru je jen do té doby, dokud je matice přidělená k procesům.

Další záležitostí je způsob jakým posílat samotné bloky. Všechny bloky jsou stejného typu jako je velká matice a implementovány jsou hlavně tyto tři: plná, řídká, skyline.

Každá z nich má jinak uchovávána data a není proto možné pro všechny napsat jedinou metodu pro přenos dat. Zde byly zavedeny identifikátory jednotlivých typů matic (předtím stačila dědičnost) a poprvé použity přepínače(tomuto řešení jsme se měli vyhýbat z důvodu, že při přidání nového typu se musí doplnit i nové možnosti do přepínačů), protože tak jak byla navržena struktura uchovávání bloků v paměti, nebylo možné na přijímající straně určit o jaký typ matice se jedná.

6 Závěr

Při bližším seznámení se Windows HPC Serverem 2008 jsme zjistili, že práce s tímto systémem je jednoduchá a intuitivní. Díky aplikaci jako je *HPC Job Manager* je práce s joby snadná, ale později se nám osvědčil příkazový řádek, pro jednoduché odesílání jobu. Díky vzdálené ploše bylo možné vyvýjet aplikace na serveru, což bylo celkem velké usnadnění.

Při psaní této práce jsme museli psát programy jak v programovacím jazyce C++ (OOSol), tak i v jazyce C# (neuronové sítě) a dle našeho názoru se MPI aplikace píší mnohem jednodušeji v C# než v C++ (Odpadá např. určovaní datového typu, který chci přenášet, není třeba ručně nastavovat velikost přenášeného pole, funkce mají méně parametrů, atd.). Na druhou stranu ale, jak je vidět v tabulce 2, cena za pohodlné psání je příliš vysoká.

Výsledky testování SOM aplikace dokazují pěknou škálovatelnost, že i když procesy komunikují mezi sebou po sítí, tak se čas se zvyšujícím počtem procesů neustále zmenšoval.

Oproti tomu u Backpropagation není zrychlení až tak zřetelné. Pokud je výpočet v rámci jednoho uzlu (není žádná síťová komunikace), tak paralelní aplikace je skutečně rychlejší. Bohužel při komunikaci po síti je to mnohem horší, ale i tak je vidět určité zrychlení (přinejmenším u neuronové sítě, které mají pouze tři vrstvy). Je to způsobeno velikou komunikací, která je nutná už z popisu algoritmu. Při paralelizaci se nám podařilo některé kroky zrušit, ale i přesto nebyl časový zisk dostatečný. Pokud bychom aplikaci psali v C++ místo C#, zrychlení by bylo určitě větší, ale nadruhou stranu pořad je tu celkem velká komunikace, takže u více vrstev by to bylo zřejmě to samé.

Na OOSolu budeme dále spolupracovat, protože se jedná o slibný projekt.

7 Reference

[1] Blaise, Barney. *Message Passing Interface (MPI)*[online]. January 3 2010.
URL. <https://computing.llnl.gov/tutorials/mpi/> (4.5.2010)

[2] Douglas, Gregor. *Debugging MPI.NET Programs in Visual Studio*[online]. September 2008.Elektronický text.
URL.<http://osl.iu.edu/research/mpi.net/documentation/Debugging%20MPI.NET%20Programs%20in%20Visual%20Studio.doc> (4.5.2010)

[3] Join, Robert. *Event Tracing with Microsoft MPI*. Jun 27 2008. Video dokument.
URL. <http://channel9.msdn.com/shows/The+HPC+Show/Event-Tracing-with-Microsoft-MPI/> (4.5.2010)

[4] Lantz, E., Haba, E. *Tracing the Execution of MPI Apllications with Windows HPC Server 2008*[online]. March 3 2009. Eletronický text.
URL.<http://resourcekit.windowshpc.net/DEVELOPER/Papers1/TracingMPIApplications.pdf> (4.5.2010)

[5] Lantz, Eric. *Windows HPC Server 2008 - Using MS-MPI*[online]. June 2008. Eletronický text.
URL. <http://www.microsoft.com/downloads/details.aspx?FamilyId=D5838485-507C-45CD-8AA6-6D6B383D1071&displaylang=en> (4.5.2010)

[6] VOLNÁ, Eva. *Neuronové sítě 1*[online]. 2. vyd. Ostrava : Ostravská univerzita v Ostravě, 2008. 86 s. Elektronický text. URL. <http://albert.osu.cz/oukip/ volna/-materialy/NEURONOVE_SITE_1/XNES1.pdf> (4.5.2010)

[7] Vondrák, Ivo. *Umělá inteligence a Neuronové sítě*. Ostrava: Ediční středisko VŠB, 1995. ISBN 80-7078-259-5

[8] *Compiling MPI Programs in Visual Studio*.
URL. <http://www.cs.utah.edu/ delisi/vsmpi/> (4.5.2010)

[9] *Kohonen's Self Organizing Feature Maps*.
URL. <http://www.ai-junkie.com/ann/som/som1.html> (4.5.2010)

[10] *MPI.NET Tutorial: Introduction*[online]. September 4 2008.
URL. <http://osl.iu.edu/research/mpi.net/documentation/tutorial/> (4.5.2010)

[11] *MPI.NET: High-Performance C# Library for Message Passing*[online]. October 6 2008.
URL:<http://osl.iu.edu/research/mpi.net/> (4.5.2010)

[12] *MPI: A Message-Passing Interface Standard Version 2.2*[online]. September 4 2009. Elektronický text.
URL. <http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf> (4.5.2010)

[13] *RFC1631 - The IP Network Address Translator (NAT)*[online]. May 1994.
URL. <http://www.faqs.org/rfcs/rfc1631.html> (4.5.2010)

[14] *Server Message Block (SMB) Version 2 Protocol Specification*[online]. May 3 2010. Eletronický text.
URL. <http://download.microsoft.com/download/a/e/6/ae6e4142-aa58-45c6-8dcf-a657e5900cd3/%5BMS-SMB2%5D.pdf> (4.5.2010)

[15] *Windows HPC Server 2008 Job Scheduler*[online]. June 2008, September 2008. Eletronický text.
URL.<http://www.microsoft.com/downloads/details.aspx?FamilyID=DAB977D5-2311-4B80-9257-477838C0EB6F&displayLang=en> (4.5.2010)

[16] *Windows HPC Server 2008 Technical Overview*[online]. June 2008, September 2008. Eletronický text.
URL:<http://www.microsoft.com/downloads/details.aspx?FamilyId=7A4544F0-81F2-4778-8A59-35C43BA49875&displaylang=en> (4.5.2010)

[17] URL. <http://www.microsoft.com/cze/sharepoint/default.aspx> (4.5.2010)

[18] URL. <http://msdn.microsoft.com/en-us/library/ms734631.aspx> (4.5.2010)

[19] URL.<http://www.microsoft.com/systemcenter/en/us/operations-manager.aspx> (4.5.2010)

[20] URL. <http://technet.microsoft.com/cs-cz/library/cc772106%28WS.10%29.aspx> (4.5.2010)

[21] URL. <http://www.microsoft.com/sqlserver/2008/en/us/> (4.5.2010)

[22] URL. <http://technet.microsoft.com/en-us/library/bb742424.aspx> (4.5.2010)

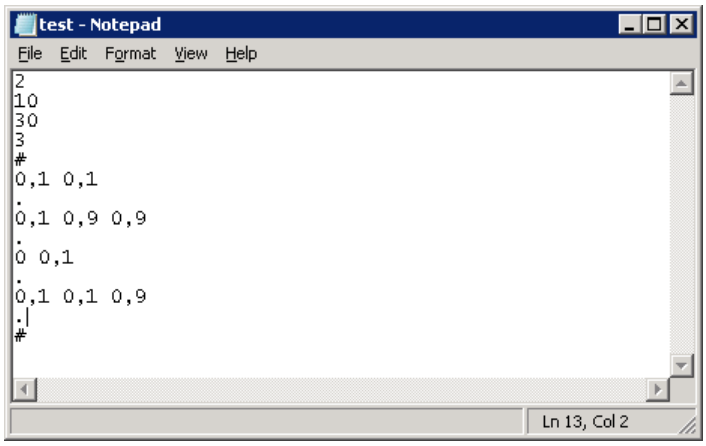
[23] URL. <http://msdn.microsoft.com/en-us/library/ms731082.aspx> (4.5.2010)

[24] URL. <http://www.am.vsb.cz/> (4.5.2010)

A Uživatelské příručky

A.1 Programu využívající Backpropagation

Samotný program, jmenuje se *NeuronSit*, se spouští s parametrem, který určuje název souboru ze kterého se budou načítat data. V tomto souboru jsou data uložena způsobem jak je ukázáno na obrázku 19. První čtyři řádky určují kolik neuronu je v každé vrstvě (první řádek jsou počet vstupů a poslední počet výstupů). Následuje oddělovač #, který určuje konec zadávání vrstev. Další data už jsou trénovací množina, která se skládá vždy ze vstupu a očekávaného výstupu. Tato data jsou vždy oddělena tečkou. K ukončení zadávání trénovací množiny je opět použit symbol #.



Obrázek 19: Vstupní data pro Backpropagation

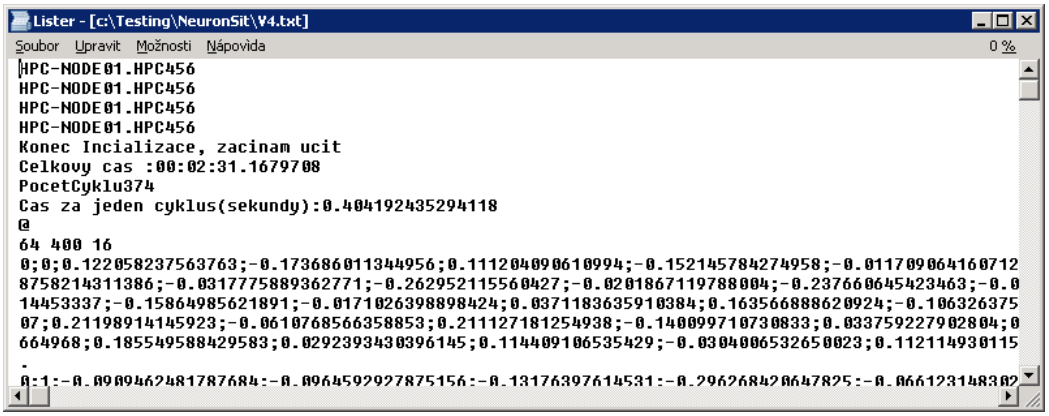
Po spuštění programu na HPC (podle návodu popsaného v podkapitole 2.2) dostaneme výstupní soubor, který obsahuje zajímavé informace o učení neuronové sítě, jako je čas, počet cyklů, chyby, uzly na kterých je výpočet spuštěn. Dále veškerá data o neuronové síti, neboli nastavení vah jednotlivých neuronů. Ukázkový výstup je na obrázku 20.

Pro ověření, že se neuronová síť skutečně naučila, jsem napsal program *TestNeuronu*, který opět při spuštění vyžaduje parametr. Tento parametr je kompletní výstupní soubor. Po spuštění programu zadáte vstupní hodnoty (oddělené mezerou) a program vypíše hodnoty jaké jsou na výstupu neuronové sítě. Pro ukončení programu zadáte hodnotu -1.

Pro generování vstupních dat, jsem si napsal generátor *Bar problému*. Tento program se jmenuje *VytvoreniZadani*. Po spuštění tohoto programu je uživatel vyzván aby zadal velikost matice a počet záznamů. Výsledek má požadovaný tvar, aby se mohl použít u backpropagation. Chybí pouze nastavení vrstev neuronové sítě. Tento výstup se uloží do souboru *data.txt*.

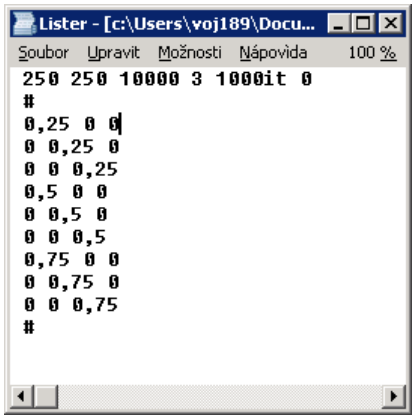
A.2 Program SOM

Tento program se nespouští na Windows HPC Serveru 2008 s parametrem, ale využívá přesměrovaného vstupu ze souboru. Jinak řečeno jeho vstupní data se zadávají pomocí



Obrázek 20: Výstupní data pro Backpropagation

console. Ukázka vstupního souboru je vidět na obrázku 21. Kde na prvním řádku z jsou hodnoty následující rozměr X, rozměr Y, počet opakování, počet vstupů, název výstupního souboru a určení jestli má být výstup zapsán do jednoho souboru (Hodnota 0 určuje, že každý proces si vytvoří vlastní soubor do kterého uloží svoji část mapy, hodnota 1 určuje, že celá neuronová síť bude uložena do jediného souboru, toto ovšem může být časově velmi náročná operace). Následuje symbol #, který oddělí nastavení neuronové sítě a vstupní hodnoty.



Obrázek 21: Vstupní data pro SOM

Pokud je výstup nastaven pro více souboru, pak se vytvoří jeden hlavní soubor, který obsahuje nejen nastavení neuronové sítě, ale i informace o učení. Poté každý proces vytvoří svůj soubor a uloží do něj data.

Data jsou v binárním souboru, tudíž je nutné pro jejich zobrazení použít program *SOMZobrazeni*. Který umožňuje zobrazení pomocí U-matrix a barevné pro tři vstupy.